Numéro d'ordre : 3462

THÈSE

présentée devant

L'UNIVERSITÉ DE RENNES 1

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention INFORMATIQUE

PAR

Monsieur Sébastien Monnet

Équipe d'accueil : projet PARIS, IRISA, Rennes

École doctorale MATISSE

Composante universitaire : IFSIC

Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données.

soutenue le 30 novembre 2006 devant la commission d'examen

Composition du jury

Monsieur Gabriel ANTONIU, chargé de recherche, INRIA	directeur de thèse
Monsieur Roberto BALDONI, professeur, Università di Roma	examinateur
Monsieur Luc BOUGÉ, professeur, ENS Cachan/Antenne de Bretagne	directeur de thèse
Monsieur Frédéric DESPREZ, directeur de recherche, INRIA	rapporteur
Monsieur Philippe PUCHERAL, Professeur, Université de Versailles	examinateur
Monsieur Pierre SENS, professeur, Université Paris 6	rapporteur

Remerciements

Trois ans c'est long, mais c'est également horriblement court ! Je souhaite ici remercier ceux qui m'ont aidé, soutenu, supporté...

Je remercie tout d'abord les six membres de mon jury pour avoir relu et évalué mon travail ainsi que pour leurs questions lors de la soutenance. Je remercie en particulier Pierre et Frédéric qui ont eu la patience de lire mon manuscrit, de rédiger des rapports détaillés et de faire des remarques qui m'ont permis d'améliorer ce document. Je souhaite également remercier Thierry pour son accueil au sein de l'équipe PARIS dont je remercie au passage tous les membres qui en font une équipe dynamique et sympathique.

Un grand merci à mes directeurs de thèse Luc et Gabriel. Gabriel merci d'avoir cru en moi dès le début et d'avoir pris le temps de m'expliquer... J'ai beaucoup appris à tes côtés. Luc merci de m'avoir fait découvrir les plaisirs du travail bien fait. Je tiens également à te remercier pour ton écoute et pour tous les conseils que tu m'as donnés sur bien des sujets...

J'ai eu la chance de faire ma thèse au sein de l'IRISA. Je voudrais remercier toutes les personnes qui en font un environnement de travail exceptionnel ! En particulier, Christiane et Élodie du service mission qui nous permettent de voyager l'esprit tranquille, merci Élodie de m'avoir sorti du "pétrin" quand j'étais à Chicago en 2004 ! Merci Maryse de t'occuper de nous comme tu le fais.

Je souhaite aussi remercier Christine Morin, Anne-Marie Kermarrec, Pierre Sens, Marin Bertier, Etienne Rivière, Christian Perez, Yvon Jegou, Françoise André, Indy Gupta, Cécile Le Pape, Landry Breuil, Loïc Cudennec et évidemment toi Mathieu ! Je vous remercie pour toute l'aide que vous m'avez apportée, pour nos discussions, vos relectures et commentaires... La liste est longue !

Je remercie également tous mes amis d'avoir compris qu'une thèse c'était une passion et demandait beaucoup de disponibilité. Je pense en particulier à Nathalie, Joris et Lise-Marie. Ces remerciements vont également à ma famille, en particulier à toi Sophie, merci pour tous tes petits messages d'encouragement sur la fin, cela m'a vraiment aidé...

Mathieu, tu m'as dit lorsque j'ai commencé ma thèse : "je suis content que tu fasses ta thèse avec nous". De mon côté, je suis ravi d'avoir fait ma thèse à tes côtés, tes qualités font de toi un bon collègue, mais aussi plus que ça ;-).

Un merci tout particulier pour Caroline. Merci simplement d'être là, de m'avoir soutenu, encouragé, aidé. Merci pour tes relectures bien sûr, mais surtout merci d'avoir accepté d'être ma femme !

Enfin je remercie mes parents. Claude, tu as toujours été là pour nous, j'ai toujours su que je pouvais compter sur toi en toute occasion, merci, merci d'être toi. Mon dernier remerciement, c'est pour toi Dominique...

Table des matières

Liste des définitions

Chapitre 1

Introduction

Sommaire

L'Homme a soif de connaître son avenir et de comprendre ce qui l'entoure. À cette fin, de nombreux instruments de mesure ont été inventés. Actuellement, ces instruments devenus très précis génèrent de grandes quantités de données. Pour mieux appréhender le monde qui l'entoure, l'Homme utilise ces données pour modéliser son environnement. En analysant les modèles et en *simulant* leur évolution, il devient possible de comprendre des phénomènes et systèmes complexes ou d'en inventer de nouveaux, et même de *prévoir* leurs possibles évolutions.

Cependant, les instruments de mesure, très précis, peuvent générer quotidiennement de grandes masses de données. Les modèles deviennent également de plus en plus complexes et prennent en compte un nombre de paramètres sans cesse grandissant. Simuler l'évolution de tels modèles demande donc une puissance de calcul et de stockage de plus en plus grande. Par exemple, en Californie, dans le cadre du projet TeraShake [127], la simulation de tremblements de terre servant à prévoir les conséquences de secousses dans cette région doit prendre en compte une masse considérable d'informations sur la topologie et la structure des terrains ainsi que sur les forces physiques entrant en jeu. Ce genre de simulation manipule de gigantesques quantités de données : au sein du projet TeraShake, les simulations prennent en entrée des centaines de gigaoctets et génèrent des résultats de l'ordre de plusieurs dizaines de téraoctets. Les chercheurs ont donc besoin d'un instrument pouvant stocker et exploiter une masse d'information considérable.

Face à cette demande croissante de puissance, les *grilles de calcul* apparaissent de plus en plus comme *la solution* de demain. En effet, ces architectures permettent d'additionner les

ressources matérielles pour former un ensemble offrant une capacité de stockage et de calcul virtuellement infinie.

Cependant une grille de calcul est un instrument particulièrement complexe car il est composé de milliers de machines souvent hétérogènes et réparties géographiquement dans des institutions qui mettent en commun leurs ressources. De plus, de nouvelles ressources peuvent être ajoutées à tout moment, de même que des ressources présentes peuvent disparaître (pannes, arrêts volontaires). Par conséquent, les concepteurs d'application s'appuient sur des *services logiciels* permettant de *simplifier* l'exploitation des grilles de calcul, d'abstraire une partie de cette complexité et de masquer partiellement l'hétérogénéité des ressources utilisées.

Les dix dernières années ont ainsi vu naître des services dédiés à certains de ces aspects. Il existe notamment des services permettant de déployer des applications sur une grille, de rechercher des ressources disponibles (matérielles ou logicielles), etc. Pourtant la conception d'applications distribuées pour les grilles de calcul reste complexe. En effet, le partage de données reste un problème difficile : si les applications sont distribuées, c'est également le cas des données auxquelles elles accèdent. Celles-ci se trouvent souvent répliquées sur plusieurs machines et le concepteur d'application doit prendre en compte ces multiples copies : il doit pouvoir les localiser, les transférer et maintenir leur cohérence alors que de multiples processus répartis dans la grille sont susceptibles de les modifier. Cette tâche est d'autant plus dure si l'on considère que les fautes et déconnexions de machines sont courantes dans les grilles de calcul et qu'elles peuvent entraîner une perte de données : si toutes les machines qui stockent une même donnée tombent en panne ou se déconnectent, la donnée n'est plus accessible.

1.1 Objectifs de la thèse

Cette thèse s'inscrit dans le cadre de la conception du service de partage de données pour la grille JUXMEM [129] (pour l'anglais *juxtaposed memory*). Ce service est développé au sein de l'équipe PARIS [123], dans le cadre du projet *Grid Data Service* (GDS [111]) de l'ACI Masse de Données, qui a débuté en 2003. JUXMEM se propose de répondre aux besoins mentionnés ci-dessus. En particulier son objectif est d'offrir les propriétés suivantes.

Persistance. Le stockage au sein de la grille doit être persistant, c'est-à-dire qu'une donnée dont la gestion est confiée à un service de gestion de données doit rester *disponible* au cours du temps. Les grilles de calcul étant des systèmes dynamiques composés de machines susceptibles de tomber en panne ou de se déconnecter, assurer la persistance des données implique la mise en place de mécanismes de tolérance aux fautes.

Localisation et transfert transparents. Les problèmes liés à la localisation des données ainsi qu'à leur transfert peuvent être traités au sein du service de gestion de données pour la grille. Ceci a pour but de simplifier le partage de données au niveau des applications. Ainsi, les applications peuvent accéder aux données partagées via des *identificateurs globaux* qui rendent transparents la localisation et le transfert des données.

Cohérence. La cohérence des données partagées répliquées doit être préservée. Ceci peut être réalisé par la mise en place de mécanismes de synchronisation afin de traiter les mises à jour concurrentes ainsi que par des mécanismes de propagations des mises à jour entre les copies d'une même donnée. Là encore, il est nécessaire de prendre

en compte la nature dynamique des grilles, les différentes copies d'une donnée étant susceptibles de disparaître à tout moment.

Dans ce contexte, cette thèse s'intéresse particulièrement aux problèmes liés à la tolérance aux fautes et à la gestion de la cohérence des données. Nous proposons des solutions pour gérer *conjointement* ces deux problèmes. La section suivante détaille nos contributions.

1.2 Contributions et publications

Architecture découplée générique. Les problèmes liés à la tolérance aux fautes et ceux liés à la gestion de la cohérence des données sont en étroite relation. Aussi avons-nous conçu une architecture en couches permettant de découpler ces deux problématiques [ADM06]. Cette architecture permet de mettre en œuvre des protocoles de cohérence indépendamment des mécanismes de tolérance aux fautes et réciproquement. De plus, sa mise en œuvre permet l'instanciation de différentes combinaisons *protocoles de cohérence/mécanismes de tolérance aux fautes*, offrant ainsi aux applications utilisatrices du service le choix de la combinaison la mieux adaptée.

Protocoles de cohérences hiérarchiques. Les grilles de calcul présentent une architecture particulière : nous verrons au chapitre suivant qu'elles ont souvent des topologies réseaux hiérarchiques. Les protocoles de cohérence classiques sont mal adaptés à de telles architectures (ils sont généralement peu performants). Nous proposons une approche hiérarchique qui permet d'adapter de nombreux protocoles de cohérence existants aux grilles de calcul [ADM06, ACM06, ACM06b].

Mécanismes de réplication hiérarchiques. Les mécanismes de réplication étudiés dans la littérature reposent souvent sur des synchronisations entre les différents acteurs or certains liens réseau des grilles de calcul présentent des latences élevées rendant ces synchronisations très coûteuses en terme de performance. De même que pour les protocoles de cohérence, nous proposons une approche hiérarchique pour les mécanismes de réplication [ADM06]. Cette approche consiste à exploiter la structure hiérarchique des réseaux des grilles pour offrir des mécanismes adaptés aux grilles de calcul.

Mécanismes de points de reprise pour les grilles. Les applications distribuées s'exécutant sur les grilles de calcul peuvent avoir des temps d'exécution longs, de l'ordre de plusieurs jours, voire plusieurs semaines. Si une ressource sur laquelle s'exécute une partie d'une application tombe en panne, il peut être intéressant de reprendre l'exécution de l'application au niveau d'un état préalablement sauvegardé, et non de la relancer complètement l'application. Nous proposons dans [MMB04a] et [MMB04b] des mécanismes hiérarchiques de sauvegarde de points de reprise pour les grilles. Ces mécanismes peuvent être employés par les applications utilisant notre service de partage de données.

Réseau logique malléable. Après avoir étudié une approche hiérarchique pour la gestion des groupes de réplication au sein des grilles, nous nous sommes intéressés à une approche permettant de viser une plus grande échelle. Cette approche, présentée en annexe, se base sur des mécanismes probabilistes offrant une maintenance plus aisée, mais des garanties restreintes. Nous avons introduit le concept de réseau logique *malléable* [MMAG06]. Il s'agit d'un réseau logique pair-à-pair qui peut s'adapter aux applications afin de mettre en place des mécanismes de réplication efficaces.

Articles dans des revues internationales

[ADM06] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation : Practice and Experience*, 18(13) :1705–1723, November 2006. Extended and revised version of [ADM04].

Chapitres de livres

[ABC+06] Gabriel Antoniu, Marin Bertier, Eddy Caron, Frédéric Desprez, Luc Bougé, Mathieu Jan, Sébastien Monnet, and Pierre Sens. GDS : An architecture proposal for a grid data-sharing service. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series, pages 133–152. Springer, 2006.

Conférences internationales avec comité de lecture

[MMAG06] Sébastien Monnet, Ramsés Morales, Gabriel Antoniu, and Indranil Gupta. MOve : Design of An Application-Malleable Overlay. In *Symposium on Reliable Distributed Systems 2006 (SRDS 2006)*, pages 355–364, Leeds, UK, October 2006.

[ACM06] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. Extending the entry consistency model to enable efficient visualization for code-coupling grid applications. In *6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 552–555, Singapore, May 2006.

[MB06] Sébastien Monnet and Marin Bertier. Using failure injection mechanisms to experiment and evaluate a grid failure detector. In *Workshop on Computational Grids and Clusters (WCGC 2006)*, Rio de Janeiro, Brazil, July 2006. Held in conjunction with VECPAR'06. Selected for publication in the post-conference book.

[ACM06b] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. A practical evaluation of a data consistency protocol for efficient visualization in grid applications. In *International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2006)*, Rio de Janeiro, Brazil, July 2006. Held in conjunction with VECPAR'06. Selected for publication in the post-conference book.

[ABJM04] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Large-scale deployment in P2P experiments using the JXTA distributed framework. In *Euro-Par 2004 : Parallel Processing*, number 3149 in Lect. Notes in Comp. Science, pages 1038–1047, Pisa, Italy, August 2004. Springer-Verlag.

[MMB04a] Sébastien Monnet, Christine Morin, and Ramamurthy Badrinath. A hierarchical checkpointing protocol for parallel applications in cluster federations. In *9th IEEE Workshop on Fault-Tolerant Parallel Distributed and Network-Centric Systems*, page 211, Santa Fe, New Mexico, April 2004. Held in conjunction with IPDPS 2004, IEEE.

[ADM04] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. Building fault-tolerant consistency protocols for an adaptive grid data-sharing service. In *Proc. ACM Workshop on Adaptive Grid Middleware (AGridM 2004)*, Antibes Juan-les-Pins, France, September 2004. Available as INRIA Research Report RR-5309.

[MMB04b] Sébastien Monnet, Christine Morin, and Ramamurthy Badrinath. Hybrid checkpoi nting for parallel applications in cluster federations. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, Chicago, IL, USA, April 2004. Poster, electronic version.

Conférences nationales avec comité de lecture

[CM05] Loïc Cudennec and Sébastien Monnet. Extension du modèle de cohérence à l'entrée pour la visualisation dans les applications de couplage de code sur grilles. In *Actes des Journées francophones sur la Cohérence des Données en Univers Réparti*, Paris, November 2005.

[DM05] Jean-François Deverge and Sébastien Monnet. Cohérence et volatilité dans un service de partage de données dans les grilles de calcul. In *Actes des Rencontres francophones du parallélisme (RenPar 16)*, pages 47–55, Le Croisic, April 2005.

1.3 Organisation du manuscrit

Dans la première partie de ce manuscrit, nous présentons le contexte de nos travaux et les travaux existants.

- Le chapitre 2 présente le cadre dans lequel se situe notre travail. Il décrit les *grilles de calcul* ainsi que les contraintes imposées par ces architectures et les *applications de couplage de codes* visées par notre service de partage de données.
- Le chapitre 3 donne la définition et la description des différents types de faute qui peuvent survenir dans les grilles de calcul. Il propose également un aperçu de l'état de l'art des mécanismes de *tolérance aux fautes*.
- Le chapitre 4 définit la *cohérence des données* et propose un tour d'horizon de la gestion de la cohérence des données dans les grappes de calculateurs, les bases de données et les systèmes pair-à-pair.

La deuxième partie présente notre contribution pour la gestion conjointe de la tolérance aux fautes et de la cohérence des données au sein d'un service de partage de données pour la grille.

- Nous commençons par une *étude de cas* au chapitre 5 afin de mettre en évidence les problématiques de tolérance aux fautes et de gestion de la cohérence des données dans les grilles. Une solution est esquissée sur un exemple.
- Le chapitre 6 décrit notre architecture logicielle permettant de gérer *conjointement* les aspects liés à la tolérance aux fautes et ceux liés à la gestion de la cohérence des données.
- Au chapitre 7, nous généralisons les idées de solutions présentées au chapitre 5 et proposons une méthode pour rendre des protocoles de cohérence existants tolérants aux fautes et adaptés aux grilles de calcul.

Dans la troisième partie, nous présentons des éléments de mise en œuvre de nos travaux ainsi qu'une évaluation des solutions proposées.

- La mise en œuvre de l'architecture présentée au chapitre 6 ainsi qu'un exemple de protocole hiérarchique tolérant aux fautes sont présentés au chapitre 8.
- Le chapitre 9 décrit une évaluation de notre contribution.

Le chapitre 10 conclut ce manuscrit et présente les perspectives offertes par nos travaux.

Enfin, une annexe propose une approche alternative permettant la gestion de données partagées en environnement volatil, à l'échelle des systèmes pair-à-pair. Il correspond à un travail effectué lors d'une collaboration et sort du cadre proprement dit du service de partage de données pour la grille.

Première partie

Contexte d'étude : gestion des données dans les grilles de calcul

Chapitre 2

Les données dans les grilles de calcul

Sommaire

Les grilles de calcul sont de plus en plus utilisées, aussi bien dans le monde de la recherche que dans celui de l'industrie. Il s'agit en effet d'un instrument puissant, permettant de traiter des problèmes difficiles faisant appel à de nombreuses données de taille importante. Cependant, la conception d'applications pour ce type de système est complexe. Les applications sont composées de processus distribués géographiquement s'exécutant sur les nœuds appartenant aux grilles. Lors de l'exécution d'une application distribuée, les processus qui la composent coopèrent notamment en partageant des données. Nous allons montrer que la gestion des données partagées entre plusieurs processus au sein d'une grille de calcul représente un facteur limitant dans la conception d'applications pour la grille. Les

données partagées doivent en effet pouvoir être localisées, transférées et modifiées de manière concurrente par de multiples processus. Cette tâche est d'autant plus complexe que les nœuds d'une grille de calcul peuvent tomber en panne ou se déconnecter.

Dans ce chapitre, nous décrivons le cadre de notre travail : le partage de données pour les applications distribuées s'exécutant sur des grilles de calcul. Dans un premier temps, nous présentons notre vision des architectures de type *grille de calcul* et en donnons une définition (section 2.1). Les besoins en terme de partage de données ne sont pas les mêmes pour tous les types d'applications distribuées. Dans notre étude, nous nous concentrons sur un type d'application distribuée particulier : les applications de couplage de codes. Nous présentons en détail ce type d'application à la section 2.2. Ensuite, nous proposons un aperçu des mécanismes actuellement utilisés pour le *partage de données* au sein des grilles et mettons en évidence leurs *limites*. Dans ce manuscrit, nous nous intéressons plus particulièrement aux problèmes liés à la gestion de la cohérence des données en présence de fautes, c'est pourquoi ces problèmes sont mis en évidence dans la section 2.4. Enfin, la section 2.5 conclut ce chapitre sur une discussion autour de la gestion de la cohérence des données partagées et la tolérance aux fautes dans les environnements de type grille.

2.1 Les grilles de calcul

L'objectif de cette section est de définir ce que nous entendons par "grille de calcul". La recherche autour des grilles de calcul est très intense, il existe de multiples approches. Nous allons ici présenter *notre* approche des grilles de calcul.

2.1.1 Historique et définition

Le mot "grille" vient de l'anglais *grid* qui a été choisi par analogie avec le système de distribution d'électricité américain (*electric power grid*), ce terme été répandu en 1998 par l'ouvrage de Ian Foster et Carl Kesselman [70]. En effet, une grille peut être vue comme un instrument qui fournit de la puissance de calcul et/ou de la capacité de stockage de la même manière que le réseau électrique fournit de la puissance électrique. La vision des inventeurs de ce terme est qu'il sera possible, à terme, de se "brancher" sur une grille informatique pour obtenir de la puissance de calcul et/ou de stockage de données sans savoir ni où ni comment cette puissance est fournie, à l'image de ce qui se passe pour l'électricité.

L'analogie avec le système de distribution d'électricité permet de cerner la vision d'une grille d'un point de vue utilisateur. Notre travail se situe au sein même des grilles, nous avons donc besoin de définir comment cette puissance est fournie.

Nous commençons par définir ce qu'est un site d'une grille, pour cela nous reprenons la définition de [74].

Définition 2.1 : *site* — Un site est un ensemble de *ressources informatiques localisées géographiquement dans une même organisation* (campus universitaire, centre de calcul, entreprise ou chez un individu) et qui forment un domaine d'administration autonome, uniforme et coordonné.

Les *ressources informatiques* sont aussi bien des liens réseau (câbles, routeurs ou *switchs*) des machines (simples PC ou calculateurs parallèles) ou des éléments logiciels. Nous pouvons maintenant définir une *grille informatique*.

Définition 2.2 : *grille informatique* — Une grille informatique mutualise un ensemble de *ressources informatiques* géographiquement distribuées dans différents sites.

Il est important de noter que les grilles informatiques sont, par nature, dynamiques : 1) les sites peuvent quitter ou rejoindre la grille à tout moment ; 2) de même, au sein de chaque site, de nouvelles ressources peuvent être ajoutées, d'autres peuvent tomber en panne ou être déconnectées.

Il existe différents types de grille informatique. On distingue notamment les *grilles de données* et les *grilles de calcul*.

Les grilles de données (*data grid*) sont principalement utilisées pour le stockage de grandes masses de données. Elles sont généralement composées de ressources offrant une grande capacité de stockage, en mémoire et sur disque.

Les grilles de calcul (*computational grid*) sont dédiées aux calculs intensifs. Une grande importance est alors donnée à la puissance des processeurs des nœuds qui la composent.

Dans ce manuscrit, bien que nous nous intéressions à la gestion de données dans les grilles, nous parlons de *grilles de calcul*. En effet, nous nous focalisons sur la gestion de données partagées par des applications scientifiques distribuées effectuant des *calculs* sur des grilles de calcul.

2.1.2 Les fédérations de grappes

Les ressources mises en commun par les sites étant souvent des grappes de calculateurs, nous allons nous concentrer sur un type particulier de grille : *les fédérations de grappes*.

Définition 2.3 : *grappe de calculateurs* — Une grappe de calculateurs est un ensemble de nœuds (typiquement des PC) interconnectés par un réseau à très faible latence et à haut débit de type SAN (*System Area Network*) ou LAN (*Local Area Network*). Généralement, les nœuds d'une grappe se situent dans une même pièce et sont reliés entre eux par des *switchs*.

Lorsque les ressources partagées par tous les sites d'une grille sont des grappes de calculateurs, on obtient une fédération de grappes. On remarque que la topologie réseau de cette architecture, représentée sur la figure 2.1, est hiérarchique : tous les nœuds d'un site sont interconnectés entre eux, alors que les différents sites sont généralement connectés par un seul lien réseau. De plus, les liens réseau reliant les nœuds d'une grappe (en traits pleins sur la figure) ont de très faibles latences[1] comparés aux liens réseau reliant les sites entre eux (en pointillés sur la figure) .

[1] Au sein d'une grappe, il n'y a pas de routeur, le seul équipement à traverser entre deux nœuds est un *switch*.

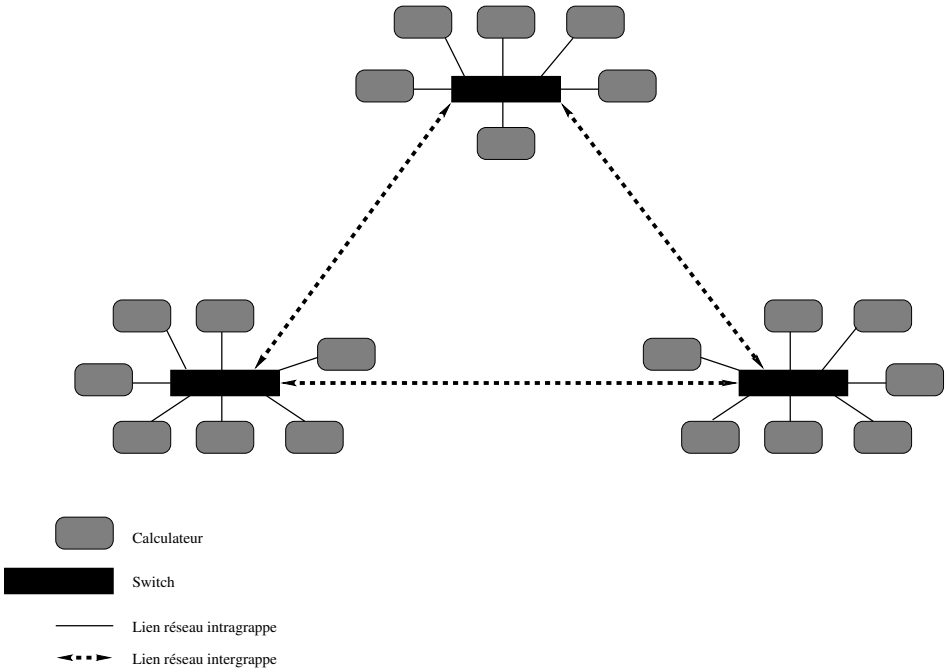

FIG. 2.1 – Un cas particulier de grilles : les fédérations de grappes.

2.1.3 Un exemple de grille : Grid'5000

En France, un important projet de grille de calcul a été lancé en 2003 : le projet Grid'5000 [113, 26]. Son but est de fédérer des grappes de calculateurs réparties dans 9 différents sites géographiques en France : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. La figure 2.2 représente les 9 sites interconnectés par le réseau Renater [125] (correspondant aux traits reliant les sites sur la carte de la figure 2.2). Ce réseau est en cours de transition[2] et va offrir un débit de 10 Gb/s pour des latences allant de 3 à 14 ms[3]. À terme, Grid'5000 devrait comporter 5000 processeurs répartis dans ces 9 sites. Au sein des sites, les nœuds sont interconnectés par des réseaux à très faible latence comme *Myrinet* [119] (pour lequel la latence entre deux nœuds est de l'ordre de 2 μs). Il y a donc un rapport de latence de l'ordre d'un facteur 10.000 entre les liens intergrappe et les liens intragrappe.

Ce type de grille de calcul correspond à ce que nous visons : c'est sur cette plate-forme que nous avons expérimenté nos contributions. Dans le monde, il existe de nombreux autres projets de grille comme TeraGrid aux États Unis [126] ou EUROGRID en Europe [109].

[2]Actuellement, certains liens offrent des débits de 10 Gb/s et d'autres des débits de 1 Gb/secondes, ces derniers sont en cours de remplacement.

[3]La variation est essentiellement due à la distance géographique.

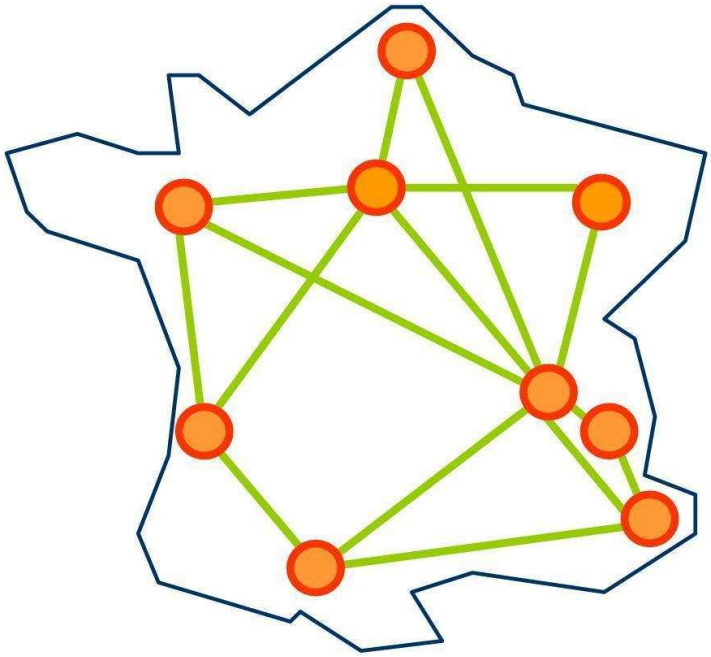

FIG. 2.2 – Carte des sites du projet Grid'5000.

2.2 Les applications de couplage de codes

Nous venons de définir les grilles de calcul et plus particulièrement les fédérations de grappes de calculateurs. Les applications s'exécutant sur ce type d'architecture sont *distribuées* : elles sont décomposées en sous-tâches, appelées processus, qui s'exécutent sur des nœuds de la grille. Les applications distribuées, composées de multiples processus, peuvent donc effectuer des tâches en parallèle : c'est ce qui permet d'obtenir une grande puissance de calcul. Lors de l'exécution de l'application, ces processus vont accéder aux données de l'application. Certaines données seront donc partagées par plusieurs processus. Nous nous concentrons sur un type particulier d'applications pour lesquelles les grilles sont bien adaptées : les *applications de couplage de codes*.

2.2.1 Description

Une application dite "de couplage de codes" est une application distribuée particulière. Elle est composée de plusieurs codes éventuellement parallèles qui coopèrent à une même fin. Les échanges entre les nœuds hébergeant une application de couplage de codes présentent par conséquent une structure hiérarchique : ils seront plus fréquents entre les nœuds hébergeant un même code qu'entre les nœuds hébergeant deux codes différents, faiblement couplés.

Les applications de couplage de codes présentent donc une topologie de partage de données hiérarchique à deux niveaux : au niveau des codes parallèles et au niveau de l'application composée des codes parallèles.

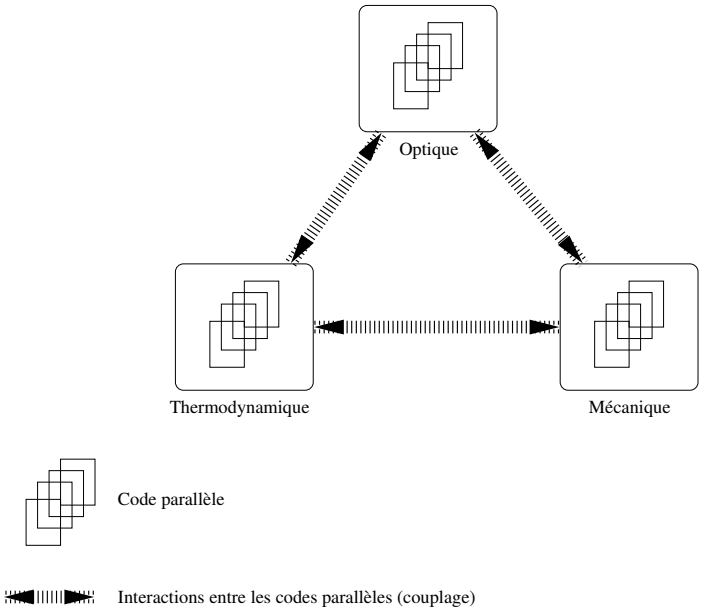

FIG. 2.3 – Une application de couplage de codes : des codes parallèles qui interagissent.

2.2.2 Exemple d'application

La figure 2.3 illustre un exemple d'application de couplage de codes. Cette application consiste en la simulation d'un satellite. Un tel objet fait intervenir plusieurs physiques telles que la thermodynamique, l'optique ou encore la mécanique. Chacune de ces physiques peut être simulée par un code parallèle au sein d'une grappe, et de temps en temps, l'état global de la simulation représentant le satellite dans son ensemble peut être mis à jour au sein des différents codes parallèles. Les applications de couplage de codes font intervenir plusieurs codes de simulation numérique (souvent parallèles) s'exécutant chacun sur une grappe de calculateurs et échangeant de temps en temps des informations (via des données partagées par exemple) afin de simuler un système complexe.

L'application *HydroGrid* [128] est un autre exemple concret. Son but est de simuler des transferts de fluides et de transport de solutés dans des milieux géologiques souterrains. Ces simulations ont pour but d'appréhender des phénomènes liés à la contamination des aquifères par des polluants, l'intrusion d'eau salée dans les aquifères ou le stockage profond de déchets nucléaires. Ces problèmes font appel à différents phénomènes physico-chimiques, chacun d'entre eux étant simulé par un code spécifique. Les différents codes, s'exécutant chacun au sein d'une institution (notamment pour des raisons de disponibilités de bibliothèques logicielles) sont couplés entre eux. Un exemple d'application représenté par la figure 2.4 simule l'intrusion d'eau salée. Elle fait intervenir deux codes parallèles, *écoulement* et *transport*, et un code séquentiel : *contrôleur*. Chacun des codes parallèles s'exécute afin de simuler l'écoulement d'une part et le transport d'autre part. De plus, comme illustré sur la figure 2.4, les deux codes parallèles s'échangent itérativement des matrices contenant des vitesses, pressions, densités et concentrations afin de simuler l'intrusion d'eau salée dans son ensemble. Le code *contrôleur* échange des données scalaires avec chacun des codes parallèles

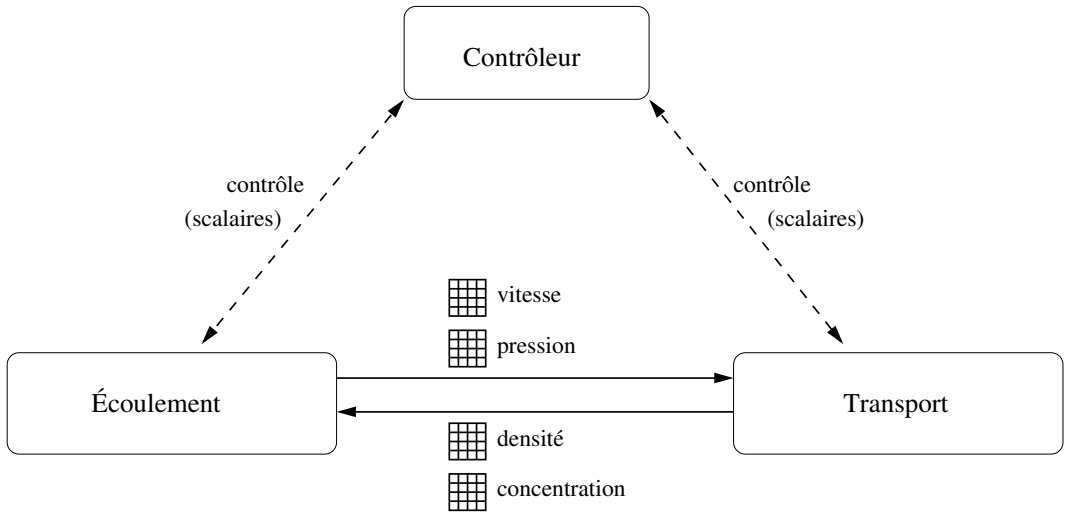

FIG. 2.4 – Échanges entre différents codes pour la simulation d'intrusion d'eau salée.

afin de piloter la simulation.

2.2.3 Les grilles de calcul : une solution pour les applications de couplage de codes

Nous avons remarqué à la section 2.1 que les grilles de calcul, et en particulier les fédérations de grappes, présentaient une topologie réseau hiérarchique avec des liens à très faible latence au sein des grappes et des liens à plus forte latence entre les grappes. Les applications de couplage de codes décrites à la section précédente présentent également une structure hiérarchique. Les codes parallèles font intervenir en général de nombreuses communications et peuvent tirer avantage d'un réseau à haute performance. Les communications entre les différents codes parallèles sont moins fréquentes et peuvent se satisfaire de plus grandes latences. De plus, il est fréquent qu'un code parallèle doive s'exécuter dans une grappe de calcul particulière. Cela peut être dû à la présence de bibliothèques logicielles disponibles au sein d'une seule grappe, ou à des contraintes d'accès à des données sécurisées placées dans une grappe particulière.

Par conséquent, les architectures de type grille de calcul sont particulièrement bien adaptées pour l'exécution d'applications de couplage de codes. En effet, en plaçant les processus d'un même code parallèle d'une application de couplage de codes au sein d'une même grappe de calculateurs[4], comme représenté sur la figure 2.5, ces processus qui partagent de nombreuses données bénéficieront des réseaux haute performance présents au sein de la grappe. Les différents codes parallèles étant *faiblement couplés*, les liens intergrappe, à plus forte latence, seront utilisés moins fréquemment.

[4]Ce positionnement des différents processus est d'autant plus naturel que les différentes institutions qui collaborent ont en général chacune une spécialité.

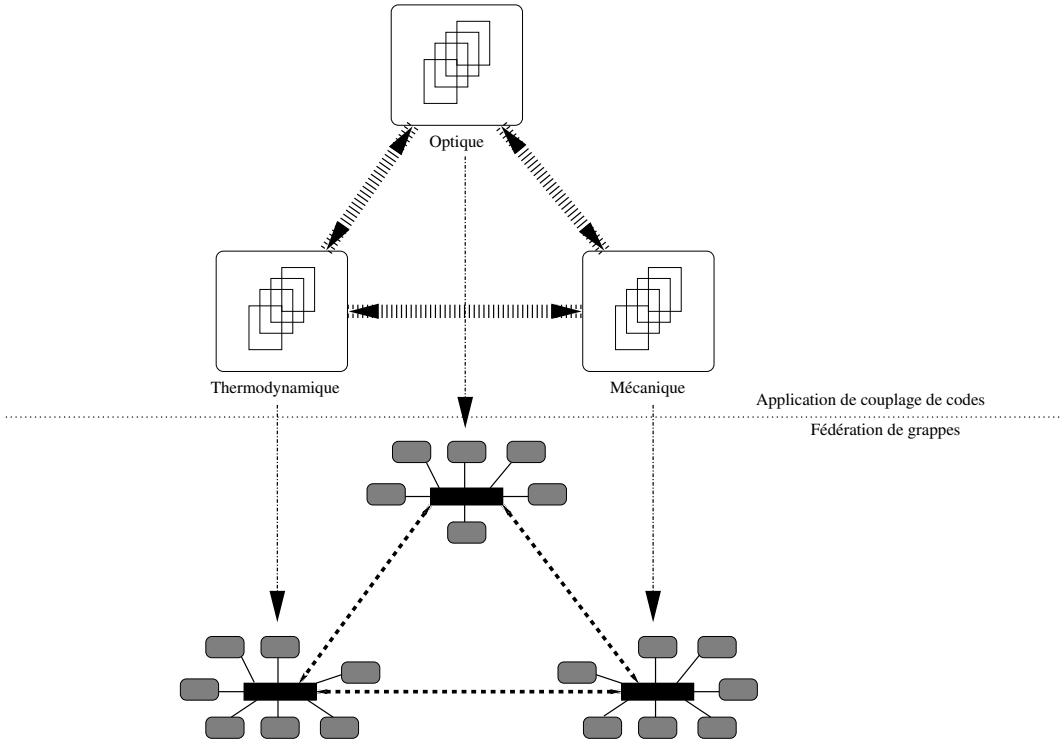

FIG. 2.5 – Les fédérations de grappes : une solution *naturelle* pour les applications de couplage de codes.

2.3 Approches pour le partage de données dans les grilles

Les applications de couplage de codes décrites à la section précédente peuvent partager des données. Nous avons vu au chapitre 1 que la quantité de données peut être très importante dans la cas d'applications comme TeraShake [127]. De plus, lorsque ces données sont partagées, elles peuvent être accédées de manière concurrente par plusieurs processus applicatifs. Nous dressons ici un panorama des systèmes existants pour le partage de données dans les grilles de calcul et nous montrons leurs limites.

2.3.1 Le partage de données dans les grilles de calcul

Dans les grilles de calcul, le partage de données est complexe. Cela est principalement dû à : 1) la grande échelle, et 2) la nature dynamique des grilles de calcul. Ces deux caractéristiques rendent impossible une connaissance globale de la plate-forme car le nombre de nœuds peut être très important, et de plus, les nœuds sont susceptibles de quitter ou de rejoindre la grille à tout instant. Cela rend difficile la localisation des nœuds hébergeant une copie d'une donnée partagée. En plus du problème de localisation de la donnée, pour lequel de nombreux travaux existent [96, 118], se pose celui du transfert de la donnée : quelle copie transférer s'il en existe plusieurs, quel protocole utiliser, quel chemin réseau emprunter, etc.

2.3.2 Systèmes existants

Depuis l'avènement des grilles de calcul, de nombreux travaux de recherche se sont intéressés aux problèmes posés par le partage des données pour les applications. Des solutions ont été proposées pour faciliter la localisation des données, pour améliorer et faciliter l'accès aux données et leur transfert, et également pour le stockage des données.

Localisation des données : utilisation de catalogues. Globus [112] est l'environnement de gestion de grille le plus utilisé. Il consiste en une collection de bibliothèques logicielles et d'outils en ligne de commande permettant d'exploiter les grilles informatiques. Pour la localisation des données, Globus s'appuie sur la notion de *catalogue*. Les catalogues sont généralement implémentés au-dessus de systèmes existants comme *Lightweight Directory Access Protocol* (LDAP [59]) ou de systèmes de gestion de bases de données. Un service de catalogues de méta-données (*Metadata Catalog Service*, MCS) est utilisé pour stocker des informations décrivant les données. Pour la localisation des copies des données répliquées, Globus propose un catalogue de copies (*Globus Replica Catalog*, GRC) et un service de localisation des copies (*Replica Location Service*, RLS [96]). Cependant, la gestion de ces catalogues reste manuelle : les applications doivent y enregistrer *explicitement* chaque nouvelle copie de la donnée. De plus, ces services n'offrent qu'une aide à la localisation et ne gèrent pas la cohérence des différentes copies. Cette solution a été utilisée par le projet européen DataGrid [44].

Il existe dans la littérature d'autres projets exploitant l'approche de catalogues de copies pour faciliter la localisation de données. On peut citer le *Metadata Catalog* (MCAT) [118] utilisé au sein du *Storage Resource Broker* (SRB) de la grille de donnée du centre de calcul de San Diego [91]. SRB offre une interface de programmation uniforme sur un ensemble de

ressources de stockage hétérogènes. Cependant, là encore, la gestion des différentes copies et de leur cohérence reste à la charge de l'application.

Dans les projets basés sur l'utilisation de catalogues, la gestion de ces catalogues n'est pas automatisée et la cohérence des données partagées n'est pas gérée. Aussi, cette solution est bien adaptée pour des données peu fréquemment modifiées, ou en lecture seule. Il existe de nombreux catalogues référençant des données *non modifiables* : BIRN [7] qui référence des données biomédicales, DPOSS [43] qui est un catalogue de données d'astronomie, GAMESS [110] qui est dédié aux chimistes, etc.

Accès aux données et transferts. Globus propose des mécanismes d'accès aux données : *Globus Access to Secondary Storage* [19] (GASS). Ce système permet aux processus applicatifs d'accéder aux données distantes de manière transparente, c'est-à-dire comme si elles étaient présentes localement sur les machines sur lesquelles ils exécutent. GASS agit donc comme un cache local. Cependant, pour accéder aux données, l'application doit appeler des primitives spécifiques (de type *ouvrir/fermer*) avant et après les accès. Ces primitives permettent au système d'effectuer les transferts de données nécessaires. Ce type de mécanisme s'apparente aux mécanismes mis en place par les protocoles de cohérence des systèmes à mémoire virtuellement partagée (MVP) que nous décrivons dans la section 4.1.

Le projet *Storage Resource Broker* (SRB [91]) se focalise également sur les accès aux données. Il a pour but d'offrir un accès uniforme à différents systèmes de stockage, qu'il s'agisse de bases de données, de systèmes d'archivage ou de systèmes de gestion de fichiers. SRB fournit une interface de programmation permettant aux applications d'accéder aux données stockées par ces différents systèmes. De plus, nous avons vu au paragraphe précédent que SRB utilise des catalogues de méta-données pour localiser les données. Cela permet d'offrir aux applications un accès transparent et uniforme à des données présentes dans des systèmes de stockage hétérogènes.

Pour le transfert des données, l'environnement Globus propose un protocole : *Grid File Transfer Protocol* [1] (GridFTP), ainsi que des bibliothèques l'implémentant. GridFTP enrichit le protocole classique *File Transfer Protocol* (FTP) afin de l'adapter aux grilles. Il offre en effet un moyen de transfert de données efficace, sécurisé et fiable. GridFTP propose des fonctionnalités telles que : 1) les transferts parallèles, afin d'augmenter le débit ; 2) des mécanismes de reprise en cas d'erreur, afin de tolérer des fautes ; et 3) des mécanismes d'authentification ou encore la vérification de l'intégrité des données, afin de fiabiliser les transferts. Ces fonctionnalités sont absentes des protocoles classiques de transfert de données comme FTP ou HTTP.

Les fonctionnalités apportées par GridFTP le rendent bien adapté aux grilles informatiques. Cependant, lors de l'utilisation d'un outil utilisant le protocole GridFTP, le déclenchement des transferts des données, ainsi que la localisation de la source et de la cible de ces transferts restent à la charge des applications. Il n'y a également aucune gestion de cohérence de données répliquées.

Stockage. Le projet *Internet Backplane Protocol* (IBP) [11] de l'université du Tennessee se focalise sur le problème du stockage de données à grande échelle (Internet). Ce projet met en œuvre un réseau de stockage sur des nœuds répartis dans le monde. Cette approche s'inspire des réseaux logistiques (*Logistical Network* [13, 12, 14]). Dans de tels réseaux, les

données transitent entre les nœuds source et destination via des nœuds intermédiaires en utilisant souvent des techniques d'ordonnancement de transfert. IBP permet ainsi de gérer un ensemble de nœuds, appelés serveurs de stockages, répartis sur Internet. Il propose une interface de programmation qui permet d'allouer des *tampons* sur un ensemble de serveurs. Les clients peuvent lire et écrire des données dans ces tampons. Ils peuvent également effectuer des transferts et des copies entre différents serveurs de stockage.

IBP n'offre que des garanties du type "au mieux" (*best effort*), à l'image d'IP (*Internet Protocol*). Comme précédemment, le déclenchement des transferts et la gestion de la cohérence des données restent à la charge des applications.

Les systèmes pair-à-pair. Certains systèmes pair-à-pair comme OceanStore [73], Ivy [85], ou Pastis [23] proposent des solutions pour le partage de fichiers modifiables à grande échelle. Ces systèmes sont décrits en détail à la section 4.2. Ils utilisent des mécanismes pair-à-pair pour localiser, transférer et stocker les données. Nous verrons au chapitre 4 que ces systèmes visent des échelles plus grandes que celles des grilles de calcul et offrent une gestion de la cohérence très relâchée.

2.3.3 Limites

Les solutions que nous venons de décrire permettent aux applications de partager des données sur les grilles de calcul. Cependant, les applications doivent gérer elles-mêmes ce partage. En effet, même si des solutions ont été proposées pour faciliter la localisation, l'accès, le transfert et le stockage des données, leur utilisation ne permet pas un partage de données modifiables transparent pour les applications. C'est en effet l'application qui doit initier *explicitement* la localisation et le transfert des données partagées.

Cette gestion explicite du partage de données par les applications pose un réel problème, surtout en ce qui concerne la gestion de la cohérence des données. Lorsqu'un processus accède à une donnée, il va la localiser, puis la transférer ou la copier sur le nœud sur lequel il s'exécute, ce qui engendre de multiples copies de la donnée. Un effet positif est de créer de multiples copies d'une même donnée, permettant ainsi des accès en parallèle sur une même donnée partagée. Mais que se passe-t-il lorsque l'une des copies d'une donnée partagée est modifiée par un processus ? Plusieurs solutions s'offrent alors au programmeur de l'application : invalider les copies rendues obsolètes par la mise à jour, propager les mises à jour en mettant en place un mécanisme de verrouillage pour prévenir les modifications qui pourraient intervenir avant la propagation des mises à jour, etc. Ces mécanismes sont difficiles à mettre en œuvre, d'autant plus dans un environnement dynamique et à grande échelle comme les grilles de calcul.

Des fautes pouvant survenir à tout moment au sein des grilles de calcul, les applications ne peuvent pas faire l'hypothèse qu'un nœud restera disponible tout au long de leur exécution pour fournir les données qu'il gère. Par conséquent, les applications doivent soit 1) s'assurer en *permanence* qu'il existe dans le système plusieurs exemplaires *à jour* de la donnée afin que celle-ci reste disponible (*réplication active*) ; soit 2) effectuer des sauvegardes régulières et être en mesure de retourner en arrière[5]. La présence de multiples copies d'une

[5]Les mécanismes de tolérance aux fautes basés sur la réplication ou sur la sauvegarde de points de reprise sont détaillés au chapitre suivant.

donnée permet de tolérer des fautes mais repose le problème de la gestion de leur cohérence.

Malgré l'aide apportée par les éléments existants décrits à la section 2.3.2, cette gestion explicite de la cohérence de données partagées en présence de fautes est un des facteurs limitant lors de la conception d'applications pour les grilles de calcul. Ce qui serait souhaitable, c'est *l'externalisation* de la gestion du partage de données afin de pouvoir n'utiliser qu'un seul et même service partagé par toutes les applications. Les concepteurs d'application pourraient alors se concentrer sur l'application elle-même en déléguant la gestion du partage de données, de la cohérence et de la tolérance aux fautes à ce service.

Il existe, notamment dans le cadre des systèmes à mémoire virtuellement partagée, des travaux permettant de partager des données de manière *transparente*, c'est-à-dire sans avoir à connaître la localisation des données, ni la façon dont elles sont transférées ou dont la cohérence est maintenue. Ces systèmes sont décrits en détail à la section 4.1. Nous verrons cependant que leur approche ne peut pas s'appliquer directement sur une architecture dynamique à grande échelle.

2.4 Problèmes induits par la dynamicité de la grille

Les grilles de calcul sont des systèmes à grande échelle, *dynamiques*. La nature dynamique des grilles est à la fois un atout, et un défi. Un atout car elle permet d'ajouter des ressources au fur et à mesure de leur disponibilité ou d'en retirer pour effectuer des opérations de maintenance par exemple. Un défi, car pour le concepteur système cette nature dynamique soulève de nombreux problèmes.

Causes. Les causes de la nature dynamique des grilles de calcul sont multiples. La première est directement liée à une caractéristique des grilles : la grande échelle. Plus le nombre de nœuds est grand, plus la probabilité d'occurrence d'une panne est importante. Par exemple, si le temps moyen interfaute (ou MTBF pour l'anglais *Mean Time Between Failures*) d'un nœud est d'une semaine et que l'on considère une grille de 10.000 nœuds, on peut prévoir qu'il y aura approximativement[6] une panne de nœud toutes les minutes ($\frac{7\,\text{jours}*24\,\text{heures}*60\,\text{minutes}}{10.000} = 1$ minute) dans le système !

Définition 2.4 : ***temps moyen interfaute (mean time between failures)*** — Le temps moyen interfaute (MTBF) est le temps moyen qui s'écoule entre deux occurrences de fautes dans un système donné.

La dynamicité peut également être induite par les administrateurs de la grille de calcul. Ils peuvent avoir à redémarrer un ou plusieurs nœuds, à en connecter de nouveaux, voire à déconnecter totalement le site de leur institution de la grille. Cette dynamicité peut avoir les mêmes conséquences que les pannes. Cependant, il est généralement possible de traiter ces cas de manière plus efficace. En effet, ces interventions peuvent être prévues et les déconnexions peuvent donc être faites "proprement" (en prévenant les autres sites ou nœuds).

[6] Approximativement car il faut alors faire l'hypothèse que les pannes sont indépendantes les unes des autres, ce qui n'est pas toujours le cas, et que chaque nœud en panne est remplacé afin de conserver 10.000 nœuds en permanence dans le système.

Notons toutefois qu'un système qui tolère les pannes tolérera également les déconnexions volontaires.

Enfin, les équipements réseaux comme des routeurs, des serveurs de noms ou des serveurs de fichiers peuvent également tomber en pannes, rendant ainsi des nœuds temporairement inaccessibles.

Conséquences. Du fait de la nature dynamique des grilles de calcul, il n'est plus possible d'avoir une connaissance globale du système car il est susceptible d'évoluer à tout moment. Cela implique de mettre en place des mécanismes de découverte et de surveillance afin d'avoir une vue approximative, partielle du système. Cette vue doit être rafraîchie pour prendre en compte les changements (ajout ou retrait de nouveaux nœuds ou de nouveaux sites).

Leslie Lamport a définit les systèmes distribués ainsi :

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Dans un cadre général, quand on prend en compte la nature dynamique d'un système distribué, on ne peut plus émettre l'hypothèse qu'un nœud puisse rester indéfiniment fonctionnel. Lorsqu'un nœud s'arrête (à cause d'une panne ou d'un arrêt volontaire), il ne joue plus son rôle dans le système. En particulier, les données qu'il stockait ne sont plus accessibles, les chemins réseau qui passaient par lui sont coupés, les nœuds qui communiquaient avec lui risquent d'être bloqués, etc. Il est donc nécessaire de prendre des précautions afin que le système puisse continuer d'évoluer malgré la dynamicité.

Il n'existe pas de solution générique pour faire face à tout degré de dynamicité. Les solutions "classiques" consistent en l'utilisation de redondance (au niveau des liens réseau afin d'avoir plusieurs routes possibles, au niveau des données afin de pouvoir conserver une copie dans le système, etc.) ou de sauvegardes périodiques. Ces mécanismes sont des mécanismes de tolérance aux fautes, certains d'entre eux sont détaillés dans ce manuscrit.

2.5 Tolérance aux fautes et gestion de la cohérence dans les grilles

Dans ce chapitre, nous avons décrit les architectures de type grille de calcul ainsi que les applications distribuées qui les exploitent. Les grilles de calculs sont particulièrement bien adaptées aux applications de type couplage de codes. Ces architectures étant dynamiques et à grande échelle, le partage de données par les applications est un problème difficile à résoudre. De plus, bien que de nombreux outils pour localiser les données ou les transférer existent, la gestion du partage et de la cohérence de ces données restent aujourd'hui à la charge de l'application.

Dans ce contexte, il semble important d'offrir aux applications la possibilité d'accéder aux données de manière transparente, c'est-à-dire sans avoir à gérer les problèmes liés à la localisation, au transfert, à la cohérence des données ou à la nature dynamique de la grille. Le chapitre suivant présente un panorama des mécanismes permettant de faire face aux problèmes liés aux fautes et le chapitre 4 un panorama sur la gestion de la cohérence des données.

Chapitre 3

Approches pour la gestion de la tolérance aux fautes

Sommaire

Dans le chapitre précédent, nous avons présenté les architectures de type grille de calcul ainsi que les applications de couplage de code qui les utilisent. Nous avons souligné que ces architectures ont une nature dynamique, c'est-à-dire que des nœuds ou des groupes de

nœuds (l'ensemble des nœuds d'une grappe par exemple), sont susceptibles de rejoindre ou de quitter la grille de calcul à tout moment.

Dans ce chapitre, nous allons nous intéresser aux approches permettant de faire face à une partie de cette nature dynamique : les approches dites de "tolérance aux fautes". La tolérance des fautes est la partie la plus difficile dans la gestion de la dynamicité[1] : il s'agit de tolérer les départs intempestifs de nœuds ou de groupes de nœuds de la grille.

Tout d'abord, nous allons décrire la notion de faute et détailler quels types de faute il est judicieux de tolérer dans les grilles de calcul (section 3.1). La tolérance aux fautes exige comme prérequis des informations sur les occurrences de fautes dans le système. Elle nécessite la mise en œuvre des mécanismes de surveillance du système que nous avons évoqués à la section 2.4 du chapitre précédent. La section 3.2 décrit les principes de la détection de fautes et met l'accent sur des détecteurs de fautes adaptés aux architectures de type grille de calcul. Les sections 3.3 et 3.4 décrivent les deux principales familles d'approches utilisées pour faire face aux fautes : les approches basées sur la réplication (section 3.3) et celles basées sur les points de reprises des applications (section 3.4). Enfin, nous concluons ce chapitre par une analyse portant sur l'adaptation de ces approches aux grilles de calcul.

3.1 Notion de faute

Il existe différents types de faute [77]. Toutes n'ont pas les mêmes conséquences et toutes ne sont pas susceptibles de se produire au sein d'une grille. Nous commençons par décrire ce qu'est une faute, puis nous donnons une classification des différents types de faute en analysant celles auxquelles il faut faire face au sein d'une grille de calcul.

3.1.1 Défaillances, erreurs et fautes

Définition 3.1 : ***défaillance (failure)*** — Un élément[2] est dit *défaillant* lorsqu'il ne se comporte pas conformément à sa spécification. Une défaillance est souvent appelée "panne".

Définition 3.2 : ***erreur (error)*** — Une erreur est un état anormal d'un élément, susceptible de causer une défaillance, mais cette dernière n'est pas encore déclarée.

Définition 3.3 : ***faute (fault)*** — Une faute peut être une action, un événement ou une circonstance qui peut entraîner une erreur.

On remarque qu'il y a des relations de causalités entre les fautes, les erreurs et les défaillances. Par exemple, pour un serveur de fichier, débrancher la carte réseau peut être une action constituant une *faute*. Cette faute a pour conséquence l'*erreur* : le serveur de fichiers n'est plus accessible. Si un client essaye d'accéder à un fichier du serveur via le réseau, cela sera impossible. Le serveur de fichier n'est donc plus conforme à sa spécification, il s'agit donc d'une *défaillance*.

La figure 3.1 illustre la chaîne de dépendances entre les fautes, les erreurs et les défaillances.

[1] Afin de gérer complètement la dynamicité, il est nécessaire de prendre en compte également les arrivées de nouveaux nœuds.

[2] Ici, un *élément* peut être un composant d'un système ou un système dans son ensemble.

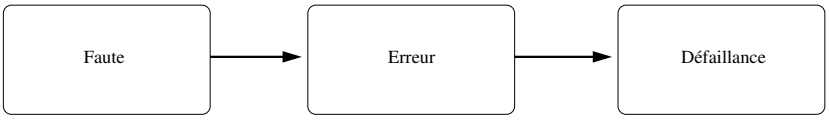

FIG. 3.1 – Dépendances entre fautes, erreurs et défaillances.

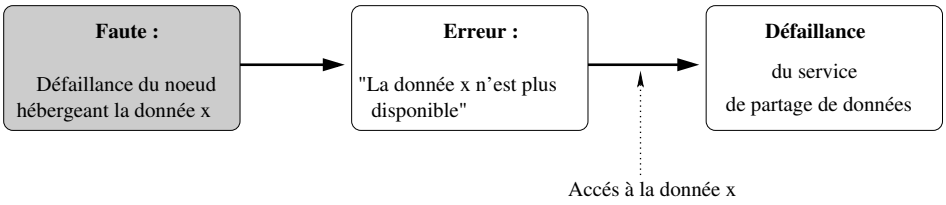

FIG. 3.2 – Exemple de défaillance du service de partage de données causée par la défaillance d'un nœud.

Par ailleurs, selon le niveau auquel un système est observé, une *défaillance* d'un élément du système peut être à l'origine d'une *erreur* d'un autre élément et donc constituer une *faute* pour cet autre élément. Par exemple, dans un service de partage de données pour la grille, la défaillance d'un nœud hébergeant la seule et unique copie d'une donnée constitue une *faute* pour le service. Cette faute entraîne une *erreur* pour le service : la donnée est alors perdue. Lorsque le service de partage de données ne satisfera pas une requête concernant la donnée perdue, le service de partage de données sera défaillant. Ceci est illustré par la figure 3.2.

Dans ce manuscrit, notre but est de tolérer les fautes qui surviennent au sein du service, ceci afin d'assurer que son comportement soit conforme à sa spécification, c'est-à-dire que le service ne soit pas défaillant. Les fautes du service étant les défaillances de ses composants, tolérer les fautes du service est équivalent à tolérer les défaillances de ses composants.

Nous parlerons donc de fautes et de défaillances selon le niveau auquel nous nous situons, mais cela correspondra toujours aux défaillances des composants (nœuds, liens réseau). Sur la figure 3.2, cela correspond à la case grisée représentant une faute du service de partage de données/une défaillance d'un nœud.

3.1.2 Types de défaillance

Les composants des grilles de calcul, les nœuds et les liens réseau, sont susceptibles d'être défaillants. Il existe plusieurs types de défaillance. Les trois principales familles sont : les défaillances franches, les défaillances par omission et les défaillances byzantines.

Définition 3.4 : ***défaillance franche ou crash (fail-stop)*** — Le composant se comporte conformément à sa spécification jusqu'à ce qu'il subisse une défaillance franche. À partir de celle-ci, il cesse définitivement toute activité.

Définition 3.5 : ***défaillance par omission (omission failure)*** — Le composant cesse momentanément son activité puis reprend son activité normale.

Typiquement, cela peut correspondre à une perte de message.

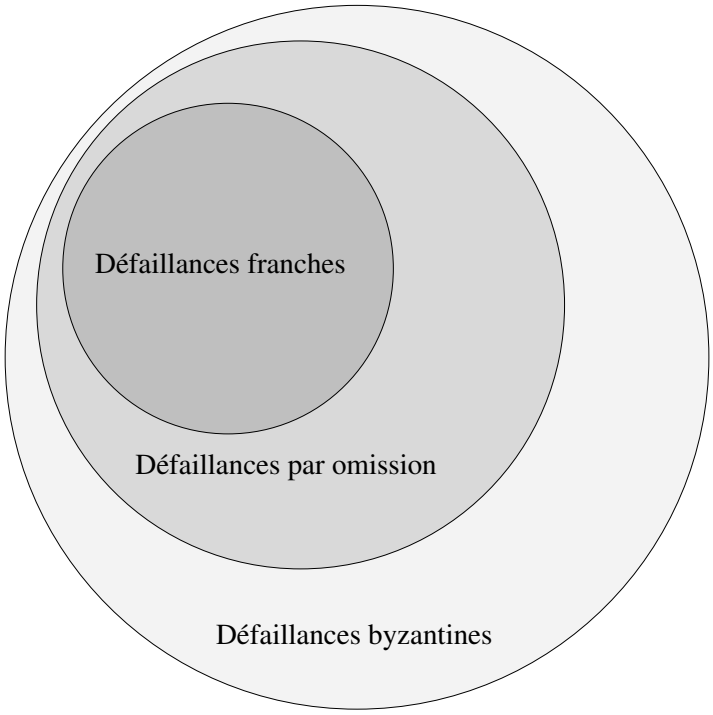

FIG. 3.3 – Imbrication des types de défaillance.

Définition 3.6 : ***défaillance byzantine (byzantine failure)*** — Un composant présentant ce type de défaillance agit de manière complètement imprévisible pour l'observateur extérieur.

C'est le type de défaillance le plus général, aucune hypothèse ne peut être faite sur le comportement d'un composant byzantin. L'étude de ce type de défaillance est très utile pour la conception d'un système sécurisé : un système tolérant un ou plusieurs composants présentant un comportement byzantin tolère alors n'importe quel comportement de la part de ces composants, notamment des comportements malicieux.

Ces différents types de défaillance sont imbriqués les uns dans les autres, comme illustré par la figure 3.3. En effet, si un système peut tolérer que des composants agissent de manière totalement imprévisible (défaillance byzantine), il peut alors également tolérer un composant qui agit selon sa spécification mais qui omet des parties de son activité (défaillance par omission). De même, la défaillance franche est un cas particulier de défaillance par omission : l'omission concerne alors tout ce qui se passe après la défaillance.

3.1.3 Quel modèle de fautes pour les grilles de calcul ?

Dans les grilles de calcul, les composants sont les nœuds et les liens réseau. Rappelons que la défaillance d'un de ses composants est une faute pour le service. Plus le type de faute à tolérer est général, plus les mécanismes de tolérance aux fautes pour y faire face sont complexes et coûteux en terme de performance : tolérer les défaillances byzantines sera ainsi plus coûteux que de tolérer les défaillances franches. Il est donc nécessaire d'évaluer

les possibilités d'occurrence de défaillances pour les différents types. En effet, il ne serait pas judicieux de mettre en place de coûteux mécanismes servant à tolérer des défaillances peu probables.

Défaillances franches dans les grilles de calcul. Le grand nombre de nœuds présents dans une grille de calcul implique une éventualité non négligeable de pannes (défaillances franches) de ces nœuds. Des pannes simultanées sont également envisageables. Lors d'une panne électrique dans une institution, toutes les ressources partagées par cette institution risquent de subir des défaillances franches simultanément, on parle alors de défaillances corrélées.

Défaillances par omission dans les grilles de calcul. La présence de certains équipements réseau tels des routeurs, notamment entre les différentes institutions composant une grille de calcul, rend possibles certaines défaillances par omission. Lorsque le réseau est congestionné, les routeurs ignorent en effet volontairement des messages pour limiter la congestion. Des pertes de messages peuvent également avoir lieu lorsqu'un nœud reçoit des paquets réseau plus vite qu'il ne peut les traiter (rappelons que les ressources d'une grille de calcul sont hétérogènes). Nous limitons donc les défaillances par omission dans les grilles aux pertes de messages. Nous considérons que les canaux de communication sont *équitables* (voir ci-dessous).

Définition 3.7 : ***canaux de communication équitables (fair lossy channels)*** — Si un processus correct p envoie un message m une infinité de fois à un processus correct q via un canal équitable, alors q reçoit ultimement (*eventually*) le message m.

Moins formellement, des messages peuvent être perdus mais, en les réémettant, ils finissent par arriver. Il est donc possible de transformer des canaux équitables en canaux *fiables* en mettant en œuvre des mécanismes de réémission.

Défaillances byzantines dans les grilles de calcul. Les grilles de calcul sont le résultat de la mise en commun des ressources d'instituts qui s'accordent mutuellement une certaine *confiance* et qui mettent en place des mécanismes coordonnés de contrôle d'accès. Les grilles de calcul sont par conséquent des environnement relativement clos et sécurisé. Nous avons choisi de ne pas traiter dans notre travail les problèmes liés à la sécurité. Si les applications qui s'exécutent sur les grilles sont correctement implémentées, il est peu probable, dans tel environnement, que des composants se comportent de manière imprévisible. Il ne semble donc pas nécessaire de mettre en œuvre des mécanismes pour tolérer les défaillances byzantines. De plus, la tolérance des défaillances byzantines requiert la mise en place de mécanismes réellement coûteux, engendrant de nombreuses communications.

Lors de la mise en place de mécanismes de tolérance aux défaillances, il est nécessaire de prendre en compte aussi bien les coûts des différents mécanismes de tolérance aux défaillances que les possibilités d'occurrence des différents types de défaillance. Au sein des grilles de calcul, il semble judicieux d'offrir des mécanismes permettant de supporter des défaillances de type *franche* ainsi que des *pertes de messages*. En effet, ces défaillances sont très courantes. En revanche, des mécanismes coûteux permettant de supporter des défaillances

byzantines ne doivent être mis en place que lorsque cela est vraiment nécessaire. Nous avons fait le choix de ne pas prendre les défaillances byzantines en compte.

3.1.4 Comment faire face aux fautes ?

Si des fautes peuvent survenir, il est alors nécessaire de les prendre en compte et de mettre en œuvre des mécanismes permettant de les gérer pour permettre aux applications de continuer leur exécution.

Il est possible de relancer une application en cas de faute. Cependant, cette technique devient de moins en moins efficace au fur et à mesure que le temps moyen entre les fautes (MTBF pour *Mean Time Between Failures*) se rapproche du temps d'exécution. Pour des applications dont la durée d'exécution est longue (en heures, en jours, voire en semaines) il devient presque impossible de terminer l'exécution.

Dans le cas d'un service pour la grille, le temps d'exécution des applications est très variable et souvent imprévisible. Le service ne doit donc pas s'arrêter à l'occurrence d'une faute et il doit rester conforme à sa spécification. Par exemple, un service offrant un stockage persistant de données doit conserver les données y compris en présence de fautes, inévitables sur une grille.

Prendre en compte les fautes de manière à maintenir les services et à permettre aux applications de s'exécuter entièrement exige la mise en place de mécanismes de surveillance du système afin de pouvoir détecter les défaillances de ses composants. La détection des défaillances est un problème complexe qui a fait l'objet de nombreux travaux de recherche, la section suivante traite de ce problème.

Pour la tolérance aux fautes, les mécanismes à mettre en place peuvent se classifier en deux grandes familles : 1) la prévention des fautes, et 2) la réaction aux fautes. Les mécanismes de prévention des fautes sont en général nécessaires au fonctionnement des mécanismes de réaction aux fautes.

Prévention des fautes. Le système est préparé à subir une ou plusieurs fautes par la création de sauvegardes (comme des points de reprise pour les applications, voir section 3.4) ou par l'utilisation de la redondance (via des mécanismes de réplication comme ceux détaillés à la section 3.3).

Réaction aux fautes. En cas de faute, le système réagit et s'adapte en s'appuyant sur les sauvegardes ou la redondance présente afin de rester opérationnel.

3.2 Détection de défaillances

La détection des défaillances dans un système est un problème complexe. Cette complexité vient de l'asynchronisme des systèmes.

Définition 3.8 : ***système asynchrone*** — Un système asynchrone est un système pour lequel il n'est fait aucune hypothèse temporelle sur les temps de transmission des messages ou sur les temps de calcul des processeurs.

Fischer, Lynch et Paterson ont prouvé [49] que dans un tel système sans hypothèse temporelle, en présence de fautes, il est impossible d'obtenir un consensus de manière déterministe entre plusieurs nœuds. En effet, lorsqu'un nœud ne communique plus avec les autres, il est impossible de déterminer s'il est défaillant ou s'il est simplement "lent" et que ses messages seront reçus plus tard. Il est donc impossible de mettre en œuvre un service de détection de défaillances *fiable* sans faire plus d'hypothèses sur le système. Dans [32], Chandra et Toueg ont proposé la notion de détecteurs *non fiables* de défaillances (*unreliable failure detectors*).

3.2.1 Principes généraux

Du point de vue de l'utilisateur, le service de détection de défaillances permet de s'abstraire des hypothèses temporelles. En effet, Chandra et Toueg ont proposé de concevoir un système sans hypothèses temporelles mais utilisant un *oracle* (un service de détection de défaillances) dont le *rôle* est de *signaler, sur chaque nœud, les nœuds "suspects" susceptibles d'être défaillants* [32]. De nombreux algorithmes, comme les algorithmes permettant d'établir un consensus entre plusieurs nœuds, peuvent s'exécuter avec de telles listes de suspects [32].

En revanche, au niveau du service de détection, il est nécessaire de faire des hypothèses temporelles. La notion de systèmes partiellement synchrones (*partial synchrony*) est décrite dans [45]. Un système partiellement synchrone est un système asynchrone vérifiant certaines hypothèses temporelles, comme l'existence de bornes temporelles sur les temps de calcul et/ou sur les temps de transmission de messages. Les systèmes réels sont rarement totalement synchrones, ni totalement asynchrones. Au sein grilles de calcul notamment, les temps de calcul ainsi que les temps de transmission des messages sont bornés, mais les bornes sont ne sont pas connues avec précision.

Fausses suspicions. Au sein des systèmes partiellement synchrones, les bornes temporelles exactes ne sont pas connues. Il n'est donc pas possible de déterminer de façon certaine la défaillance d'un nœud. C'est pourquoi les détecteurs de défaillances sont dits "non-fiables". Ils ne fournissent pas des listes de nœuds défaillants, mais des listes de nœuds *suspectés* d'être défaillants. Il est donc possible qu'un nœud défaillant ne soit pas suspecté durant un certain temps ou qu'un nœud non-défaillant soit suspecté à tort. Ce dernier cas s'appelle une *fausse suspicion*.

Pour le système utilisateur du service de détection, les fausses suspicions peuvent avoir de lourdes conséquences : il faut alors gérer la présence d'un nœud que l'on a considéré pendant un certain temps comme en panne. Le nœud faussement suspecté n'est plus à jour par rapport au reste du système qui l'avait considéré comme absent. Il est donc nécessaire dans ces cas-là de réparer le système, ce qui peut s'avérer délicat selon le rôle que joue le nœud suspecté.

3.2.2 Classification des détecteurs de défaillances

La mesure de la qualité d'un détecteur de défaillances dépend de l'utilisation qui en est faite. En effet, tous les détecteurs de défaillances ne sont pas comparables. Les deux prin-

cipaux critères de qualité pour des détecteurs de défaillances sont : 1) la *complétude*, et 2) la *justesse*.

Définition 3.9 : ***complétude (completeness)*** — Un nœud défaillant doit être détecté comme étant défaillant.

Définition 3.10 : ***justesse (accuracy)*** — Un nœud non-défaillant (correct) ne doit pas être considéré comme étant défaillant.

La complétude se dérive en deux catégories : 1) *forte* : ultimement[3], tous les processus corrects suspecteront tous les nœuds défaillants de façon permanente ; 2) *faible* : ultimement, chaque nœud défaillant est suspecté par au moins un processus correct.

De même, la justesse se dérive en quatre catégories : 1) *forte*, les nœuds corrects ne sont jamais suspectés (c'est-à-dire qu'il n'y a pas de fausses suspicions) ; 2) *faible*, il existe au moins un nœud correct qui n'est jamais suspecté ; 3) *ultimement forte* ; et 4) *ultimement faible*.

Ces critères permettent de répartir les détecteurs de défaillances en 8 grandes classes résumées par le tableau ci-dessous. Ces différentes classes de détecteurs ne sont pas toutes comparables entre elles. Selon la classe visée, la réalisation d'un détecteur de défaillance nécessite : 1) un plus ou moins grand nombre de messages, et 2) des hypothèses sur le synchronisme du système plus ou moins fortes, par exemple la conception d'un détecteur de défaillance *parfait* (P) impose de formuler de fortes hypothèses de synchronisme sur le système.

	Justesse forte	Justesse faible	Justesse ultimement forte	Justesse ultimement faible
Complétude forte	Parfait P	Fort S	Ultimement parfait $\Diamond P$	Ultimement fort $\Diamond S$
Complétude faible	Q	Faible W	$\Diamond Q$	Ultimement faible $\Diamond W$

Sur le plan pratique, [36] propose d'évaluer les détecteurs de défaillances en prenant également en compte le temps de détection ainsi que le nombre et la durée des fausses suspicions.

3.2.3 Mise en œuvre de détecteurs de défaillances

Il existe deux grandes techniques pour la mise en œuvre de détecteurs de défaillances : celle basée sur des échanges périodiques de *messages de vie*, et celle basée sur des allers-retours "*ping/pong*".

Échanges de messages de vie (*heartbeats*). Chaque nœud envoie périodiquement un message de vie à *tous* les autres et attend donc, à chaque période, un message de vie de chacun d'entre eux. Lorsqu'un nœud ne reçoit pas de message de vie d'un autre nœud, il le considère comme *suspect*. De nombreux projets de recherche se sont concentrés

[3]Ici ultimement signifie "il existe un moment à partir duquel..." (*eventually*).

sur la fiabilisation de la suspicion, en prenant en compte le nombre de fausses suspicions ou la variation de latence dans l'arrivée des messages de vie d'un nœud particulier [36, 42, 62, 55]. Cette technique a été employée pour mettre en œuvre les détecteurs de défaillances présentés dans [17]. Le principal inconvénient de cette technique réside dans l'utilisation de communications de type "tous-vers-tous", qui ne sont pas adaptées à l'échelle des grilles de calcul, en raison du grand nombre de nœuds.

Messages "ping/pong". De manière périodique ou à la demande, les nœuds envoient un message "ping" à tous les autres nœuds ou à une partie d'entre eux. Cette technique peut permettre une détection plus ciblée : elle permet de ne surveiller qu'un sous-ensemble des nœuds. En revanche, pour obtenir autant d'informations qu'avec la technique d'échange de messages de vie, deux fois plus de messages sont nécessaires (chaque message de vie "pong" étant réclamé explicitement par un message "ping"). Le système de composition de groupe à grande échelle SWIM (pour *Scalable Weakly-consistent Infection-style process group Membership* [41]) utilise ce type de mécanisme pour détecter les nœuds défaillants.

3.2.4 Passage à l'échelle

Les détecteurs de défaillances imposant que chaque nœud surveille directement tous les autres nœuds ne sont pas adaptés à l'échelle des grilles de calcul. Nous distinguons trois approches pour le passage à l'échelle des détecteurs de défaillances : l'approche hiérarchique, l'approche basée sur un anneau logique et l'approche probabiliste.

Approche hiérarchique. Les auteurs de [17] proposent une version hiérarchique de détecteurs de défaillances utilisant des échanges de messages de vie [18, 16]. L'ensemble des nœuds est partitionné en sous-ensembles et les échanges de messages de vie de type "tous-vers-tous" sont restreints à chaque sous-ensemble. Chaque sous-ensemble choisit un représentant et les différents représentants des sous-ensembles s'échangent des messages de vie. Cette approche permet d'obtenir des détecteurs de défaillances adaptés aux systèmes à grande échelle comme les grilles.

Approche basée sur un anneau logique. L'utilisation d'un anneau logique déterminant les schémas d'échanges de messages de vie et de propagation de l'information est proposée par [78]. De même que pour l'approche hiérarchique, cela permet de limiter le nombre de messages de vie émis sur le réseau et d'offrir de bonnes propriétés de passage à l'échelle.

Approche probabiliste. Une autre approche permettant de concevoir des détecteurs de défaillances pour les systèmes à grande échelle est l'utilisation de techniques probabilistes. Dans SWIM [41], les nœuds envoient des messages de type "ping" périodiquement à un ensemble *aléatoire* de nœuds. De plus, les messages "ping" et "pong" contiennent les informations sur les nœuds suspects permettant ainsi de propager les détections.

Les mécanismes probabilistes n'utilisent pas de notion de représentant ni de topologie spécifique. Leur maintenance est donc plus aisée, ce qui les rend bien adaptés aux systèmes à *très grande échelle*. En revanche, les défaillances sont détectées avec une forte probabilité mais non avec certitude. À l'opposé, les détecteurs basés sur une approche hiérarchique

ou s'appuyant sur une topologie en anneau offrent une bonne réactivité. De plus les détecteurs hiérarchiques présentés dans [18, 16] s'adaptent bien à la topologie des fédérations de grappes. Ils nous semblent donc les mieux adaptés aux grilles de calcul.

3.3 Techniques de réplication

La détection des défaillances est un prérequis pour la tolérance aux fautes. En cas de faute, le système doit avoir prévu des moyens de permettre aux services de continuer son fonctionnement. La réplication consiste à conserver en permanence dans le système plusieurs copies. Ainsi, en cas de fautes, les copies restantes peuvent continuer à assurer le service.

Pour mettre en place des mécanismes de réplication, il est nécessaire de pouvoir : 1) gérer des groupes de nœuds qui hébergent les différentes copies, et 2) mettre en place des communications de groupe afin d'assurer les mises à jour des différentes copies. Ces deux points sont liés car il est possible de gérer des groupes à l'aide de mécanismes de communication de groupe.

3.3.1 Gestion des groupes de copies

La gestion des groupes de copies se fait par des *protocoles de composition de groupe* aussi appelés *protocoles d'appartenance* (*group membership* en anglais). Le rôle de ces protocoles est de gérer des listes de membres de groupes. Ils permettent à de nouveaux nœuds de rejoindre un groupe, ou à des nœuds membres d'un groupe de le quitter. Ils peuvent utiliser des mécanismes de détection de défaillances afin d'évincer un nœud défaillant de son groupe.

Au sein d'un groupe, chaque nœud possède une *liste des membres* de ce groupe. Un protocole de composition de groupe fait évoluer ces listes au gré des arrivées et départs de nœuds. Ces listes doivent rester cohérentes entre elles, c'est pourquoi le protocole "harmonise" les listes des différents membres, on parle de "vue" du groupe. Cette harmonisation est rendue nécessaire par la dynamicité des groupes de copies. Un départ (volontaire ou dû à une défaillance) ou une arrivée de nœud doit être reflété sur chacun des autres membres, cela correspond à un changement de "vue". De nombreux travaux de recherche [37, 54, 30, 82] associent donc la composition de groupes aux problèmes d'accord dans les systèmes distribués. En effet, le problème de composition de groupe peut être résolu en mettant en place des mécanismes permettant aux membres de se mettre d'accord sur la composition de leur groupe. Un protocole de composition de groupe basé sur des mécanismes d'accord génériques est décrit dans [54]. Une *pile de protocoles* est proposée par [82]. Au sein de cette pile, un protocole de consensus est utilisé pour gérer la composition des groupes.

Une notion de composition de groupe existe également dans le domaine des systèmes pair-à-pair [41]. Il s'agit alors de construire des réseaux logiques au-dessus des réseaux physiques, en reliant les pairs appartenant à un même groupe. Cependant, la gestion de ces groupes est faite "au mieux" (*best effort*) et n'offre en général pas les garanties qui seront nécessaires à la gestion de données modifiables partagées par des applications scientifiques, par exemple la possibilité de réaliser une diffusion efficace et fiable.

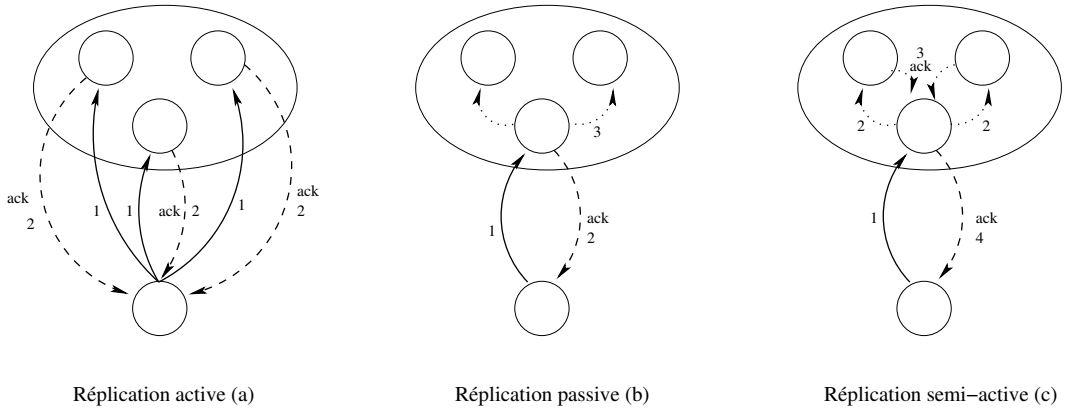

FIG. 3.4 – Différents types de mécanisme de réplication.

3.3.2 Propagation des mises à jour

Lorsqu'une entité est répliquée, la mise à jour d'une copie doit être propagée aux autres copies. Il est donc nécessaire de mettre en œuvre des mécanismes de communication de groupe au sein des groupes de copies. Selon la fréquence des mises à jour, des hypothèses sur la fiabilité du système et des contraintes sur les garanties de fraîcheur des copies, les besoins en terme de communication de groupe sont très variables. Aussi, existe-t-il de nombreuses approches permettant d'appliquer des mises à jour à un ensemble de copies. On distingue notamment la *réplication active*, la *réplication passive* et la *réplication semi-active*. Certains systèmes (comme DARX [81], une plate-forme pour des applications multi-agent) proposent différents mécanismes de réplication afin de pouvoir s'adapter au cas d'utilisation. La figure 3.4 illustre ces différents types de mécanisme de réplication.

Réplication active. Toutes les copies reçoivent et traitent les messages de manière concurrente (partie a de la figure 3.4). Avec ce type de réplication, les mises à jour sont envoyées à toutes les copies par le nœud effectuant la mise à jour. Ce dernier attend ensuite des acquittements des copies mises à jour. Cette technique implique que pour communiquer avec un groupe il est nécessaire de connaître (ou de pouvoir localiser) l'ensemble des membres du groupe.

Réplication passive. La réplication passive (partie b de la figure 3.4) est une technique optimiste car seule une copie est mise à jour. Depuis cette copie (appelée copie primaire) les mises à jour sont propagées périodiquement aux autres membres du groupe. Pour communiquer avec le groupe il suffit alors de communiquer avec la copie primaire. La mise à jour peut être perdue si le nœud hébergeant la copie primaire subit une défaillance avant de la transmettre aux copies de sauvegarde.

Réplication semi-active. Dans le cas de la réplication semi-active (partie c de la figure 3.4), les mises à jour ne sont envoyées qu'à une copie. Cette dernière met à jour l'ensemble du groupe avant de retourner un acquittement au nœud initiateur de la mise à jour. Les garanties offertes ici sont les mêmes qu'avec la réplication active mais il suffit de connaître la copie primaire du groupe pour pouvoir envoyer une mise à jour. C'est en revanche la technique qui induit le plus de latence lors des mises à jour (4 phases de communication contre 2 pour les autres types de réplication).

Mises à jour concurrentes. Dans le cadre général, plusieurs nœuds peuvent envoyer des mises à jour à un groupe de copies de manière concurrente. Afin de conserver la cohérence des différentes copies, il est nécessaire que ces mises à jour soient appliquées dans le même ordre sur chacune des copies. Ceci peut être réalisé à l'aide de mécanismes de diffusion atomique.

Définition 3.11 : ***diffusion atomique*** — La propriété de diffusion atomique au sein d'un groupe assure que chaque membre du groupe prend en considération les mêmes messages, dans le même ordre.

De nombreuses approches pour concevoir des protocoles de communication de groupe offrant la propriété de diffusion atomique sont présentées dans [46]. La diffusion atomiques peut être réalisée par l'utilisation d'un nœud *sérialisateur* par lequel toutes les mises à jour transitent. Ce nœud joue alors un rôle critique et doit pouvoir être remplacer en cas de défaillance. Une approche alternative [82, 53] consiste à utiliser des mécanismes d'accord au sein des groupes pour déterminer l'ordre dans lequel les messages doivent être pris en compte. Dans ce cas, des échanges de messages supplémentaires permettent aux nœuds récepteurs de la diffusion de s'accorder sur l'ordre de prise en compte des messages diffusés.

3.3.3 Utilisation de groupes de copies

Les groupes de copies (et donc la réplication) sont utilisés dans de nombreux systèmes [81, 105, 20, 101], afin de fiabiliser des données, des fichiers ou encore des processus.

La plate-forme pour applications multi-agent DARX [81] utilise la réplication afin de fiabiliser des agents logiciels. Ce système propose différents mécanismes de réplication (active, passive et semi-active). Il propose d'adapter le mécanisme de réplication ainsi que le nombre de copies en fonction : 1) des agents répliqués, et 2) de l'estimation du niveau de risque de fautes.

Le système de fichier LegionFS [105] repose sur la réplication offerte dans Legion [34] afin de proposer un système de fichiers fiable dont le but est de permettre la collaboration à grande échelle. Les données des fichiers sont répliquées afin d'assurer leur persistance au sein du système. Les mécanismes de réplication utilisés peuvent être spécifiés par l'utilisateur.

Horus [101] et Isis [20] sont des systèmes offrant des mécanismes de communication de groupe permettant de mettre en place des mécanismes de réplication. De nombreuses applications sont construites au-dessus de ces systèmes. Par exemple, des applications financières et de courtage utilisent le système Isis.

Les chapitres 6 et 7 détaillent notre utilisation de la réplication afin de fiabiliser un service de partage de données.

3.4 Sauvegarde de points de reprise

Les approches basées sur la réplication sont bien adaptées à la tolérance aux fautes pour préserver des données. Elles nécessitent en revanche l'utilisation d'un grand nombre de

nœuds et sont difficiles à mettre en place lorsqu'il s'agit de répliquer des processus applicatifs communicants et non nécessairement déterministes. Dans ce cas, des solutions basées sur des sauvegardes de points de reprise semblent plus judicieuses.

3.4.1 Principe de base

Un point de reprise pour un processus applicatif est un ensemble d'éléments (l'état de la mémoire par exemple) permettant de relancer ce processus en cas de défaillance d'un nœud. Il peut être établi de manière incrémentale : seules les données modifiées depuis le dernier point de reprise sont alors sauvegardées. Certains systèmes permettent la duplication de processus (*fork*), ce qui autorise la création du point de reprise en arrière-plan, sans bloquer le processus.

Dans le cas d'une application distribuée composée de plusieurs processus, un point de reprise est composé d'un ensemble de points de reprise des processus de l'application. Les communications interprocessus entraînent alors des dépendances. Afin d'exprimer ces dépendances, Leslie Lamport a défini la relation de causalité entre deux événements *(happened before)* [76]. Au sein d'un même processus, un événement en précède un autre s'il se produit avant dans le temps. Pour deux processus communicants, l'émission d'un message par un processus précède la réception de ce message par son destinataire. Par ailleurs, la relation de causalité est transitive. Une exécution répartie peut donc être vue comme un treillis d'événements. La principale difficulté rencontrée lors de la mise en place de mécanismes de points de reprise pour une application distribuée consiste à trouver un ensemble valide d'états, c'est-à-dire *sans dépendance* entre les états qui le composent. Cet ensemble est appelé un *état global cohérent*.

Le calcul d'un état global cohérent a donné lieu à de nombreuses recherches. Une solution élégante a été proposée par K.M. Chandy et L. Lamport [33]. Cette solution utilise un système de "marqueurs" envoyés par le nœud initiateur du calcul de l'état global cohérent puis retransmis par les nœuds le recevant (après avoir sauvegardé leur état local). En supposant les canaux de communication *FIFO* (*First In First Out*, "premier entré, premier sorti"), les marqueurs permettent aux processus de journaliser les messages reçus des nœuds n'ayant pas encore transmis de marqueur. Cette technique ajoute cependant des données aux messages lors de la création d'un point de reprise. De plus, elle nécessite la journalisation de messages reçus. En effet, dans l'état restauré, certains messages ont été envoyés mais non reçus. Le processus destinataire doit donc les "rejouer". Le système de tolérance aux fautes décrit dans [93] utilise un procédé semblable à celui-ci. Dans ce dernier, les processus confirment le point de reprise au processus initiateur qui envoie une confirmation à tous les processus lorsque la définition du point de reprise est complète.

Dans les solutions proposées pour les modèles de sauvegarde/reprise, on distingue deux grandes familles. La première famille, pessimiste, consiste à calculer un état global cohérent lors de la sauvegarde du point de reprise. Ce type de technique est dit "coordonné". En effet, cela consiste à synchroniser les processus lors de la sauvegarde. Les protocoles de la deuxième famille, optimiste, ne calculent l'état global cohérent que lors de la reprise après une défaillance. Les points de reprise sont sauvegardés pour chacun des processus indépendamment les uns des autres. Cela évite le surcoût dû à la synchronisation mais rend les techniques de reprise plus complexes. La grande diversité des solutions proposées s'explique par l'hétérogénéité des systèmes qui vont les exploiter : par exemple, certaines sont conçues pour

des systèmes à très grande échelle [22], dans lesquels une synchronisation des processus serait très coûteuse (voir impraticable), d'autres sont adaptées aux grappes de calculateurs à petite échelle utilisant un réseau à très haut débit et faible latence [100, 9, 39, 93, 69, 84].

3.4.2 Points de reprise coordonnés

Établir un point de reprise coordonné consiste à synchroniser les processus lors de la sauvegarde des états locaux afin de garantir que l'état global obtenu à l'aide de l'ensemble de ces états locaux soit cohérent. Une *ligne de recouvrement* est ainsi créée lors de l'établissement du point de reprise. De ce fait, l'application peut être relancée en relançant chacun des processus (ou seulement les fautifs si un système de journalisation est mis en place) au niveau de son dernier point de reprise : aucun autre calcul supplémentaire n'est nécessaire. On remarque qu'il suffit alors de conserver le dernier point de reprise de chaque processus. En effet, tous les états sauvegardés antérieurement deviennent obsolètes. En revanche cette technique présente un inconvénient majeur : elle nécessite la synchronisation de tous les processus, ce qui engendre un surcoût qui peut être élevé, surtout dans des systèmes à grande échelle.

Il existe de nombreuses améliorations possibles pour limiter le surcoût dû à la synchronisation des processus [83, 47]. En premier lieu, il n'est pas nécessaire que *tous* les processus se synchronisent. En effet, seuls les processus ayant des dépendances doivent le faire. Koo et Toueg [71] proposent un algorithme ne synchronisant qu'un nombre minimal de processus. Ensuite, la quantité de données à sauvegarder peut être grandement diminuée en utilisant une technique de points de reprise incrémentale. Les auteurs de [93] proposent d'utiliser les services de GENESIS, un système pour les grappes de PC fournissant un modèle de mémoire partagée, afin de ne bloquer les processus que si cela est nécessaire (c'est-à-dire si leur exécution n'entraîne pas de dépendances, comme l'envoi de messages). Une autre optimisation consiste à profiter des barrières de synchronisation de l'application pour effectuer les points de reprise. Des mécanismes implémentés sur TreadMarks [3] sauvegardent des points de reprise coordonnés (entre autres), et profitent de la synchronisation du système lors du lancement d'un "ramasse miettes" pour effectuer ce type de point de reprise [39]. Une solution apportée dans [84] pour les systèmes à mémoire virtuellement partagée exploite la réplication des données inhérente à ces systèmes pour la sauvegarde des points de reprise. La présence de répliques sur différents sites permet de minimiser les transferts de données. De plus, le fait que les données soient conservées dans la mémoire de plusieurs nœuds et non sur disque permet des accès rapides aux données de reprise. De surcroît, les données utiles pour la tolérance aux fautes peuvent être exploitées par le système de mémoire partagée ce qui permet de compenser en partie le surcoût dû aux mécanismes de tolérance aux fautes.

3.4.3 Points de reprise non-coordonnés

Points de reprise indépendants. Contrairement aux points de reprise coordonnés, les points de reprise indépendants ne nécessitent pas de synchronisation interprocessus au moment de la sauvegarde des points de reprise. Les processus sauvegardent leur état de temps en temps de manière indépendante, le surcoût dû à la synchronisation est ainsi évité. De plus, chaque processus peut ainsi choisir un moment propice à la sauvegarde de son état,

par exemple quand il y a peu de données à sauvegarder. Ceci rend cette technique intéressante lors des exécutions sans faute. Cependant, les états sauvegardés n'appartenant pas nécessairement à un état global cohérent, le nombre d'états à conserver pour la reprise peut être très important introduisant ainsi un surcoût en terme d'espace de stockage. La mise en place d'un "ramasse-miettes" est également nécessaire pour supprimer les états devenus inutiles. De plus, afin d'éviter des successions de retours en arrière, appelées *effet domino*, des mécanismes de journalisation doivent être mis en place, apportant également un surcoût y compris lors des exécutions sans faute. Enfin, la reprise après faute est complexifiée car il faut calculer la ligne de recouvrement, c'est-à-dire jusqu'où les processus doivent revenir en arrière afin de redémarrer sur un état global cohérent.

Points de reprise induits par les communications. La technique des points de reprise induits par les communications peut être vue comme une variante de celle des points de reprise indépendants [47]. Il s'agit de sauvegarder des points de reprise indépendants comme décrit ci-dessus, mais, afin de limiter l'ampleur du retour en arrière potentiel, les processus sont contraints de temps en temps de sauvegarder leur état. En effet, si l'on force un processus à sauvegarder son état à chaque réception de message, cela élimine l'effet domino. Cependant, cet avantage est acquis au prix de nombreux points de reprise inutiles. Des solutions permettent de limiter ce nombre grâce à l'utilisation de données supplémentaires portées par les messages de l'application, par exemple des graphes ou parties de graphes de dépendances.

Alors que les mécanismes de réplication vus à la section 3.3 sont bien adaptées pour améliorer la disponibilité des données, les mécanismes de sauvegarde de points de reprises sont plus adaptés pour la tolérance aux fautes des applications. Nous proposons à la section 7.4 des mécanismes de sauvegarde de points de reprises hiérarchiques hybrides, alliant des sauvegardes de points de reprises coordonnés au sein des sites des grilles et des sauvegardes de points de reprise induits par les communications entre les sites. Ces mécanismes peuvent être mis en place afin de tolérer les défaillances des clients du service de partage de données.

3.5 Tolérance aux fautes dans les grilles : vers une approche hiérarchique

La nature dynamique des grilles de calcul est en grande partie due à la présence de fautes. Ces fautes sont essentiellement les défaillances franches des nœuds qui la composent. Il existe de nombreux travaux de recherche, aussi bien sur la détection que sur la tolérance de ce type de défaillance. Nous avons notamment vu des approches basées sur des techniques de réplication et sur des mécanismes de points de reprise. Cependant, ces études ont souvent été menées de manière théorique ou dans des systèmes distribués à petite échelle. De plus, à l'exception de [18], ces études ne prennent généralement pas en compte la topologie réseau des grilles de calcul. Aussi, de nombreuses solutions proposées reposent-elles sur des synchronisations entre groupes de nœuds. Dès lors que l'on souhaite adapter ce type de solution aux grilles de calcul, il semble judicieux de prendre en compte la topologie du réseau utilisé. En effet la différence de latence, d'un facteur 10.000, entre les réseaux internes des grappes de calculateurs et celle des réseaux externes reliant ces grappes entre elles rend les

synchronisations intragrappe bien moins coûteuses que les synchronisations faisant intervenir des nœuds localisés dans des grappes différentes. Le chapitre 7 décrit notre approche hiérarchique pour la gestion de la tolérance aux fautes, prenant en compte cette différence.

Chapitre 4

Approches pour la gestion de la cohérence de données

Sommaire

Les applications distribuées permettent de faire des opérations en parallèle sur plusieurs nœuds. Pour ce faire, les données partagées accédées par différents nœuds se retrouvent souvent dupliquées afin de permettre des accès en parallèle et de limiter le nombre de transferts. Dans le chapitre 2, nous avons montré que cette gestion de la cohérence devenait un facteur limitant lors de la conception d'applications pour les grilles de calcul. Nous proposons donc d'externaliser cette gestion vers de notre service de partage de données. Ainsi, toute application qui utilisera notre service de partage de données en bénéficiera.

Dans ce chapitre, nous nous focalisons sur la gestion de la cohérence de données. Nous présentons dans la section 4.1 les approches qui ont été conçues pour les systèmes à mémoire virtuellement partagée (MVP). Ces approches présentent l'intérêt d'offrir des modèles de cohérence efficaces, adaptés au calcul à haute performance et donc aux applications que nous visons, les applications de couplage de codes décrites à la section 2.2. En revanche, elles ne prennent généralement en compte ni les aspects liés à la grande échelle, ni ceux liés à la présence de fautes. Des solutions prenant en compte ces aspects ont été développées dans le cadre des systèmes pair-à-pair (P2P) décrits dans la section 4.2. Enfin, nous décrirons la gestion de la cohérence au sein des systèmes de bases de données répliquées dans la section 4.3.

4.1 Modèles et protocoles de cohérence dans les systèmes à mémoire virtuellement partagée

Les *systèmes à mémoire virtuellement partagée* ont été conçus pour des architectures de type *grappe de calculateurs*. Ils ont pour but d'offrir l'illusion d'une mémoire unique avec un seul espace d'adressage. Les systèmes à mémoire virtuellement partagée permettent ainsi de programmer des applications distribuées pour des grappes de calculateurs de la même manière que l'on programme les calculateurs multi-processeurs à mémoire partagée. La gestion de la mémoire est totalement déléguée au système et les programmeurs d'applications accèdent à la mémoire sans avoir à localiser ou à transférer explicitement les données.

4.1.1 Notion de cohérence

Un système à mémoire virtuellement partagée est responsable de la gestion de la cohérence des données. Le système propose aux programmeurs un *modèle de cohérence*. Le modèle de cohérence décrit la manière dont la cohérence des différentes copies des données est gérée. C'est le modèle de cohérence qui va déterminer comment les mises à jour d'une donnée par un processus seront visibles par les autres processus.

Les *modèles de cohérence* sont mis en œuvre par des *protocoles de cohérence*. Il existe de nombreux modèles de cohérence. Tous n'offrent pas les même performances et n'imposent pas les mêmes contraintes aux programmeurs d'applications. Nous distinguons deux grandes familles de modèles[1] : les modèles à cohérence *forte* et les modèles à cohérence *relâchée* (ou *faible*). Cette classification discrimine les modèles de cohérence en fonction des contraintes qu'ils imposent aux programmeurs : les modèles de cohérence imposant aux applications l'utilisation d'opérations autres que les opérations d'accès aux données (comme des opérations de synchronisation) sont des modèles de cohérence relâchée, ils sont également appelés modèles de cohérence *avec* synchronisation, les autres sont des modèles de cohérence forte, ou modèles de cohérence *sans* synchronisation.

[1] Il existe d'autres manières de classifier les modèles de cohérence, celle choisie correspond à notre vision.

4.1.2 Modèles de cohérence forte

Les modèles de cohérence forte n'impliquent pas l'utilisation d'opérations spécifiques. Les applications accèdent aux données et le système gère les transferts et copies de manière complètement transparente. Les modèles de cohérence forte se distinguent les uns des autres par les garanties offertes en terme de cohérence de données.

Cohérence stricte (*atomic consistency*). La cohérence stricte correspond à l'intuition naturelle de la notion de cohérence. Dans ce modèle, toute lecture retournera la dernière valeur qui a été écrite. C'est le modèle de cohérence qui est généralement mis en œuvre au niveau des unités de gestion mémoire dans les systèmes mono-processeurs. La mise en œuvre d'un tel modèle au sein d'une MVP s'avère cependant très coûteuse : lors de chaque accès à la mémoire partagée, une synchronisation globale est nécessaire afin de satisfaire les garanties.

Cohérence séquentielle (*sequential consistency*). La cohérence séquentielle a été formalisée par Leslie Lamport en 1979 [75]. Ce modèle est moins restrictif, il garantit que chaque processus "voit" toutes les opérations dans le même ordre, mais il ne garantit pas qu'une lecture retournera la dernière valeur affectée par une écriture. Avec ce modèle de cohérence, le résultat de toutes les exécutions est le même que si les opérations de tous les processus avaient été exécutées dans un ordre séquentiel donné dans lequel toutes les opérations de chaque processus suivent l'ordre du programme. Les premiers systèmes à mémoire virtuellement partagée utilisent ce modèle de cohérence.

Cohérence causale (*causal consistency*). Le modèle de cohérence causale [60] relâche les contraintes par rapport au modèle de cohérence séquentiel. Il n'est en effet pas toujours nécessaire d'obtenir un ordre total unique vu par chacun des processus. Ce modèle se base sur la relation de causalité (*happened before*) décrite dans [76] pour déterminer l'ordre des écritures. La relation *happened before* permet de lier entre eux certains événements par un ordre partiel bien défini et de relâcher les contraintes sur les événements indépendants. La plus grande partie des applications tolèrent en effet que deux écritures indépendantes ne soient pas vues dans le même ordre par tous les processus.

4.1.3 Modèles de cohérence relâchée

Les modèles de cohérence relâchée imposent aux programmeurs l'utilisation de primitives supplémentaires, généralement des primitives de synchronisation. Les informations fournies par ces primitives permettent aux protocoles de cohérence implémentant ces modèles d'être plus efficaces en diminuant le nombre de messages sur le réseau. Aussi, ces modèles sont souvent utilisés par les applications de calcul scientifique à haute performance.

Nous présentons ici différents modèles de cohérence relâchée.

Cohérence faible (*weak consistency*). Ce modèle propose deux types d'accès : les accès *ordinaires* (lectures/écritures) et les accès *de synchronisation* (accès aux objets de synchronisation). Les écritures ne sont pas propagées lors des accès ordinaires. En revanche, lors des accès de synchronisation, toutes les informations sont mises à jour. Le modèle garantit que l'ensemble de la mémoire est cohérent à chaque point de synchronisation, c'est-à-dire : 1) que les toutes les modification locales sont propagées aux autres processus ; et 2) que toutes les modifications des autres processus sont visibles localement.

Cohérence à la libération (*release consistency*). Le modèle de cohérence à la libération améliore le modèle de cohérence faible en relâchant les garanties de cohérence : contrairement au modèle de cohérence faible, *l'ensemble de la mémoire* n'est pas nécessairement cohérent à chaque opération de synchronisation. Avec ce modèle, deux types d'accès de synchronisation sont distingués : *acquire* et *release*. Ces deux opérations permettent de délimiter une section critique au sein de laquelle une partie de la mémoire partagée sera accédée. Ainsi, lors d'un accès de type *acquire* placé en entrée de section critique, le modèle garantit que toute les écritures des autres processus sont visibles localement ; et lors d'un accès de type *release* en sortie de section critique, le modèle garantit que toutes les écritures locales sont propagées aux autres processus.

Cohérence à la libération paresseuse (*lazy release consistency*). La cohérence à la libération paresseuse relâche encore plus les contraintes et réduit encore le nombre de communications. Le modèle de cohérence à la libération propage toutes les écritures locales lors des accès synchronisés de type *release*. Or certaines de ces propagations sont inutiles car les processus concernés n'effectuerons pas nécessairement des accès aux données mises à jour. La cohérence à la libération paresseuse réduit encore le nombre de communications en ne propageant les écritures qu'aux processus qui déclarent leur intention d'accéder aux données, en effectuant un accès synchronisé de type *acquire*. Cette cohérence a été implémentée dans le système à MVP TreadMarks [3]. Une implémentation consiste à confier la gestion des accès aux données à un nœud particulier. Cette technique s'appelle : "*home-based lazy release consistency*" (HLRC) [107].

Cohérence à l'entrée (*entry consistency*). Le système à MVP Midway [15] a introduit le modèle de cohérence à l'entrée. Ce modèle associe à chaque variable partagée un objet de synchronisation spécifique. Cette association variable/objet de synchronisation permet de ne transférer que les mises à jour des variables nécessaires, et non plus de toute la mémoire. De plus, il est alors possible d'avoir des écritures concurrentes sur des variables différentes. Le modèle de cohérence à l'entrée introduit également une nouvelle opération de synchronisation : *acquire_read*. Le modèle différencie ainsi les accès en mode exclusif (*acquire*) des accès en mode non-exclusif (*acquire_read*). Cette distinction permet d'autoriser des lectures concurrentes d'une même variable. Cependant, les lectures concurrentes à des écritures ne sont pas possibles.

Cohérence de portée (*scope consistency*). Le modèles de cohérence de portée [61] est proche du modèle de cohérence à l'entrée. Le modèle de cohérence à l'entrée impose au programmeur d'associer explicitement un verrou à chaque donnée. Le modèle de cohérence de portée quant à lui utilise les opérations de synchronisation déjà présentes dans le programme. Dans ce cas, l'association verrou/donnée est réalisée implicitement par analyse du programme.

4.1.4 Approches pour la localisation des données

Un protocole de cohérence mettant en œuvre un modèle de cohérence doit pouvoir localiser les données. La localisation d'une donnée revient à trouver le nœud stockant la copie la plus à jour[2], c'est-à-dire celui hébergeant le processus ayant effectué la dernière écriture. Ce nœud est appelé *propriétaire* de la donnée. On appelle nœud *gestionnaire* d'une donnée le

[2]Ou satisfait un critère de fraîcheur propre au modèle de cohérence.

nœud qui est chargé de localiser son propriétaire. On distingue des gestionnaires *distribués* et des gestionnaires *centralisés*.

4.1.4.1 Gestionnaires centralisés

Dans le cas d'un gestionnaire centralisé, toutes les requêtes d'accès à une donnée, les lectures comme les écritures, sont adressées au nœud gestionnaire, choisi. Ce nœud est responsable de la gestion de l'ensemble des données.

Cette gestion centralisée est bien adaptée à de petites architectures possédant des réseaux d'interconnexion performants comme les grappes de calculateurs. En revanche, elle présente des limites en terme de passage à l'échelle. Au sein des systèmes à grande échelle, les latences des communications entre les nœuds peuvent être grandes. Lorsque des nœuds "éloignés" en terme de latence du nœud gestionnaire accèdent à une donnée partagée, les échanges réseau entre ces différents nœuds risquent de ralentir l'application dans son ensemble.

En plus des limites liées au passage à l'échelle, ces protocoles sont sensibles aux fautes. Si le nœud gestionnaire tombe en panne, l'ensemble des données partagées devient inaccessible.

4.1.4.2 Gestionnaires distribués

Les approches basées sur un gestionnaire distribué permettent une décentralisation du gestionnaire. Nous distinguons trois types de gestionnaire distribué : 1) *gestionnaire distribué fixe*, 2) *gestionnaire distribué à diffusion*, et 3) *gestionnaire distribué dynamique*.

Gestionnaire distribué fixe. Cette approche permet un meilleur équilibrage de charge grâce à l'utilisation de plusieurs nœuds gestionnaires. Chaque nœud gestionnaire est responsable d'un sous-ensemble des données partagées. Cette approche ne résout cependant en rien les problèmes liés au passage à l'échelle et à la tolérance aux fautes mentionnés ci-dessus.

Gestionnaire distribué à diffusion. Cette approche consiste à diffuser les requêtes de localisation à tous les nœuds du système. Le gestionnaire se trouve donc distribué sur l'ensemble des nœuds. Ceci implique de nombreuses diffusions globales, ce qui est très coûteux en terme de communication. Cette approche est conçue pour des systèmes de petite taille et devient impraticable dès que la taille du système augmente. En revanche la localisation des données reste possible en cas de fautes.

Gestionnaire distribué dynamique. Cette approche a été proposée par Li et Hudak [79]. Elle reprend le principe de celle du gestionnaire distribué fixe, à ceci près que tous les nœuds gèrent la localisation de toutes les données. Cependant, cette localisation n'est pas exacte : les nœuds connaissent seulement un propriétaire *probable*. En cas de changement de propriétaire d'une donnée lors d'un accès en écriture, l'ancien propriétaire transmet la donnée au nouveau et le considère comme le nouveau propriétaire probable. Il pourra ainsi lui retransmettre les requêtes d'accès à la donnée. Ce mécanisme permet de mettre en place un *chaînage* afin de pouvoir systématiquement retrouver la donnée. Ces mécanismes sophistiqués permettent une localisation efficace des données au sein des systèmes à MVP s'exécutant sur des grappes de calculateurs. En revanche, là encore, les problèmes liés au passage à l'échelle ainsi qu'à la tolérance aux fautes

ne sont pas résolus. Au contraire, le mécanisme de chaînage accentue des problèmes. Le chemin réseau à emprunter pour suivre le chaînage peut exploiter des liens à forte latence. De plus, la localisation d'une donnée devient impossible dès qu'un maillon de la chaîne, c'est-à-dire un nœud, est en panne.

4.2 Modèles et protocoles de cohérence dans les systèmes pair-à-pair

Les protocoles de cohérence pour les MVP, conçus pour des grappes de calculateurs, font implicitement l'hypothèse que la taille du système est restreinte (quelques centaines de nœuds au maximum) et que les fautes sont exceptionnelles. Il est vrai cependant que certains systèmes à mémoire virtuellement partagée prennent en compte les défaillances franches de nœuds [69, 67, 9]. En revanche, les systèmes pair-à-pair ont démontré d'excellentes propriétés de passage à l'échelle et de tolérance aux fautes. Des systèmes comme KaZaA [116], Gnutella [86] ou eDonkey [108] réunissent des millions de nœuds qui se connectent et se déconnectent avec une fréquence élevée.

Cependant, la majeure partie de ces systèmes [116, 86, 108] est dédiée à des applications de partage de fichiers et ne gère que des données *non modifiables*, c'est-à-dire en lecture seule. Il s'agit généralement d'une gestion "au mieux" (*best effort* en anglais). Ils ne proposent par conséquent aucun support pour la gestion de la cohérence des données. Récemment, quelques systèmes pair-à-pair proposant la gestion de données modifiables ont vu le jour [73, 85, 23, 40]. Nous les décrivons brièvement dans cette section.

4.2.1 Les systèmes pair-à-pair

Les systèmes pair-à-pair sont par nature complètement distribués. Dans ces systèmes, chaque pair peut être à la fois client et serveur. Plus généralement, chaque pair peut même jouer l'ensemble des rôles existants dans le système. Il en résulte que les communications au sein de ces systèmes sont bien distribuées et qu'ils sont très bien adaptés aux grandes échelles. De plus, ces systèmes présentent une bonne tolérance à la volatilité (connexions et déconnexions de nœuds).

Afin de pouvoir s'exécuter à très grande échelle, les systèmes pair-à-pair reposent sur la notion de *réseau logique* (*overlay* en anglais). Un réseau logique pair-à-pair définit les "liaisons" logiques entre les différents pairs du système, c'est-à-dire quel pair "connaît" quel pair. Ceci permet à chaque pair de n'avoir à gérer qu'une vision *partielle* du système, et non le système dans sa totalité. C'est principalement cette vision partielle qui confère aux systèmes pair-à-pair leurs propriétés de passage à l'échelle et de tolérance à la dynamicité.

Les systèmes pair-à-pair sont généralement classés en deux familles en fonction du réseau logique qu'ils utilisent : nous distinguons les systèmes pair-à-pair *non-structurés* des systèmes pair-à-pair *structurés*.

Systèmes pair-à-pair non-structurés. Au sein des systèmes pair-à-pair non-structurés, chaque pair connaît un sous-ensemble de pairs. Ce sous-ensemble est généralement déterminé de manière aléatoire mais il existe cependant de nombreux travaux de

recherche visant à rapprocher au sein du réseau logique les pairs ayant des affinités [102, 95, 57, 103] ou à prendre en compte la distance en latence entre les pairs [41]. Ces réseaux logiques permettent en général de supporter une grande dynamicité. Gnutella [86] est basé sur un tel réseau.

Systèmes pair-à-pair structurés. Les systèmes pair-à-pair structurés s'appuient sur une structure distribuée, en général une table de hachage (DHT pour l'anglais *Distributed Hash Table*). Chaque pair se voit attribuer un identifiant résultat de l'application d'une fonction de hachage sur son adresse IP (par exemple SHA1 ou MD5). Les pairs peuvent ainsi être classés selon leur identifiant au sein d'une table de hachage distribuée. Chaque pair possède alors un *index* contenant un certain nombre d'autres pairs répartis sur la DHT. Le contenu de cet index varie selon l'implémentation. En utilisant cette approche, des systèmes pair-à-pair comme Chord [99], Pastry [94] ou Tapestry [106] obtiennent une couche de routage très performante.

Systèmes pair-à-pair hybrides. Certains systèmes pair-à-pair s'inspirent à la fois des systèmes structurés et des systèmes non-structurés. Des systèmes comme KaZaA [116] utilisent la notion de "super-pair" : les pairs "classiques" sont liés à des super-pairs qui eux-même sont reliés entre eux. Les implémentations actuelles de la spécification JXTA [115] proposent une table de hachage distribuée entre les super-pairs appelés *pairs de rendez-vous*.

4.2.2 Approches pour la localisation des données

Les systèmes pair-à-pair utilisent le réseau logique sous-jacent pour la localisation des données et le routage.

Les systèmes non-structurés utilisent des mécanismes d'inondation pour localiser les données. Pour cela, un pair envoie une requête à chacun de ses voisins dans le réseau logique puis chacun des voisins la retransmet à chacun de ses voisins, et ainsi de suite. Des mécanismes sont généralement mis en place pour limiter l'inondation : le nombre de retransmissions est limité ; de plus, à chaque retransmission, le nombre de voisins à contacter peut être réduit grâce à des filtres de Bloom. Enfin de nombreux travaux de recherche visent à accélérer la localisation des données conservant en cache les résultats des recherches antérieures. Les mécanismes de recherche des systèmes non-structurés offrent une localisation *incertaine* : des données présentes dans le système peuvent ne pas être trouvées lors d'une recherche.

Les systèmes structurés attribuent des identifiants aux données dans le même espace que celui des identifiants des nœuds. Le pair dont l'identifiant est numériquement le plus proche de celui d'une donnée est *responsable* de cette donnée. Cela signifie qu'il connaît l'emplacement de cette donnée, à l'image des nœuds gestionnaires des MVP. Ainsi, la localisation d'une donnée revient à acheminer une requête vers le pair dont l'identifiant est le plus proche. La DHT au sein de laquelle sont répartis les pair permet de le faire de manière efficace, généralement en $\log(n)$ sauts logiques[3], où n est le nombre de pairs dans le système. De surcroît, la localisation dans ces systèmes est *certaine*.

[3]Un saut logique correspond au chemin physique séparant deux voisins dans le réseau logique.

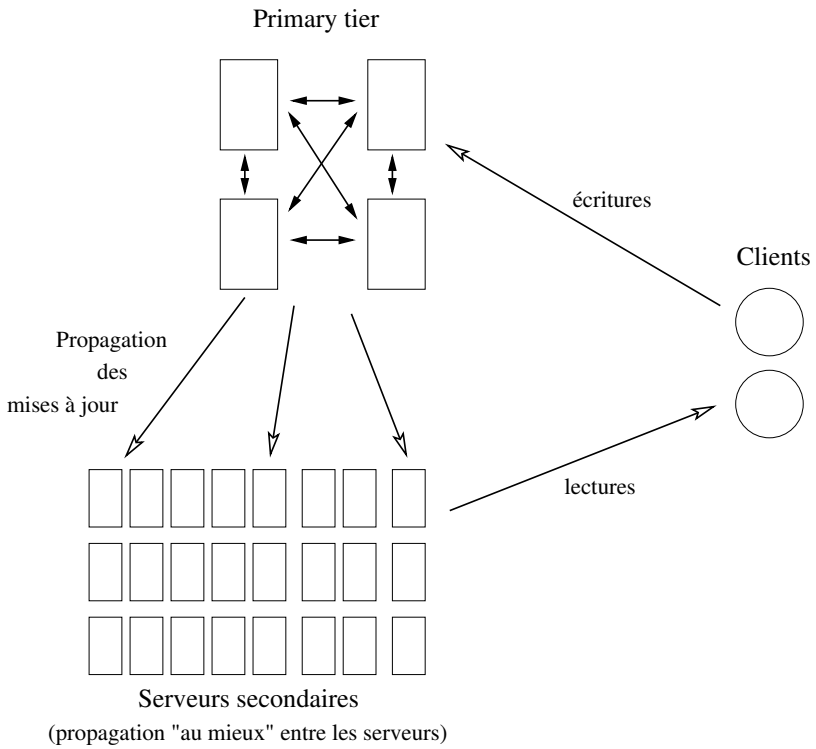

FIG. 4.1 – Communications au sein du systèmes Oceanstore.

4.2.3 Cohérence des données dans les systèmes pair-à-pair

La majorité des systèmes pair-à-pair sont conçus pour partager des fichiers en lecture seule [116, 86, 108]. Dans ce cas-là, les problèmes de cohérence des données sont inexistants : les données sont placées et répliquées dans le système puis ne sont jamais modifiées. La principale problématique à laquelle font face ces systèmes consiste donc en la recherche de données présentes dans le système.

Toutefois, certains systèmes pair-à-pair comme OceanStore [73], Ivy [85] et Pastis [23] s'intéressent au partage de données modifiables. Ce sont trois systèmes de gestion de fichiers.

OceanStore [73] est illustré sur la figure 4.1. Il permet de partager des données modifiables. Ce système utilise un ensemble de pairs jouant un rôle particulier appelé *primary tier*. Il est responsable de la gestion des modifications apportées aux données. C'est le *primary tier* qui reçoit les requêtes de modification. Il est en charge de la résolution des conflits en décidant de l'ordre d'application des modifications. Cette décision est ensuite transmise à l'ensemble des pairs possédant une copie. L'ensemble de pairs composant le *primary tier* est supposé stable et possédant un bon réseau d'interconnexion. Ce système se rapproche donc du modèle client/serveur où le serveur est mis en œuvre par un ensemble de pairs particuliers.

Ivy [85] est un système exploitant la notion de journaux (*logs*) pour permettre à plusieurs nœuds d'effectuer des écritures concurrentes. Comme l'illustre la figure 4.2, chaque

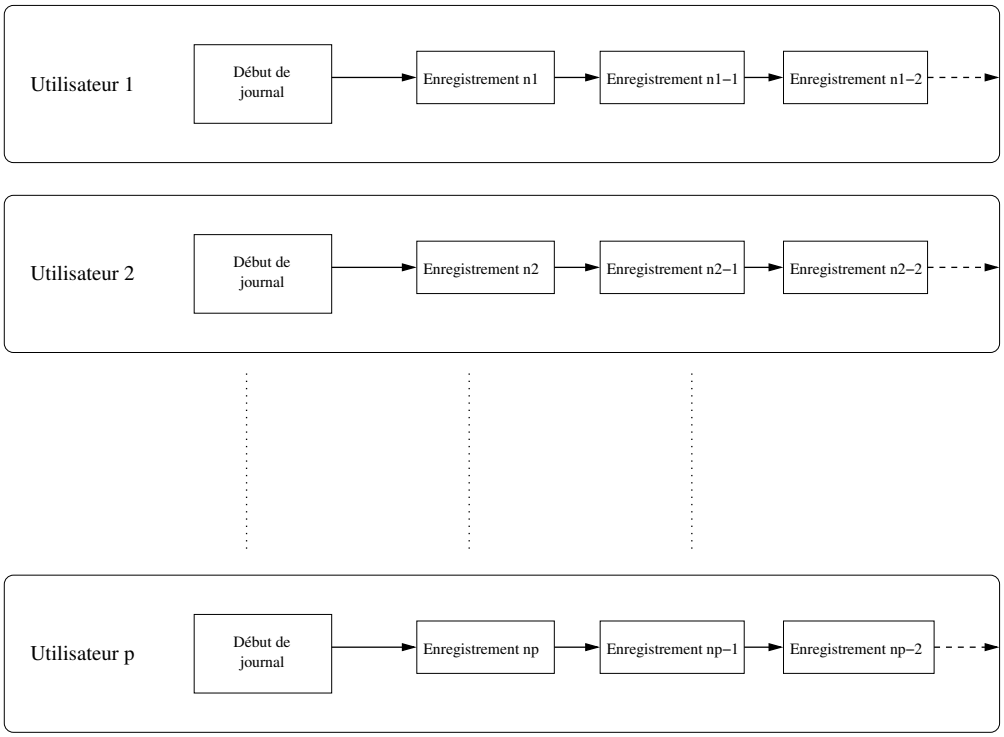

FIG. 4.2 – Ivy utilise un journal par utilisateur pour stocker les écritures.

utilisateur possède un journal dans lequel il écrit les modifications apportées à chaque donnée partagée. Ivy, basé sur Chord, utilise la DHT pour stocker les journaux des différents utilisateurs. La lecture d'un fichier nécessite donc le parcours de multiples journaux. De plus, dès qu'un fichier est partagé par plusieurs utilisateurs, des conflits peuvent apparaître. Ivy propose des outils permettant leur résolution manuelle. Le modèle de cohérence implémenté par ce système est de type relâché.

Pastis [23] est conçu comme un véritable système de gestion de fichiers. Il s'appuie sur la notion de descripteur de fichiers (en anglais *i-node*). Les descripteurs de fichiers sont stockés dans une DHT (PAST) sous forme de blocs modifiables ; ces blocs sont appelés UCB pour l'anglais *User Certificate Block*. Ils contiennent en effet des données permettant de vérifier l'intégrité du bloc ainsi que son propriétaire. Les descripteurs de fichiers contiennent des pointeurs sur des blocs de contenu : les CHB pour l'anglais *Content Hash Block*. Les CHB sont des blocs non-modifiables. Les modifications de fichiers entraînent donc des suppressions/ajouts de CHB et des modifications d'UCB. La figure 4.3 illustre les relations entre ces différents blocs. Le système Pastis met en œuvre des mécanismes complexes de signature afin d'assurer l'intégrité et l'authenticité des fichiers. Il propose deux modèles de cohérence relâchée : 1) le modèle *close-to-open*, dans lequel que les écritures ne sont propagées qu'après la fermeture du fichier et les écritures distantes ne sont visibles qu'à l'ouverture du fichier ; et 2) le modèle plus relâché *read your writes* qui garantit que chaque lecture reflète toutes les écritures locales précédentes.

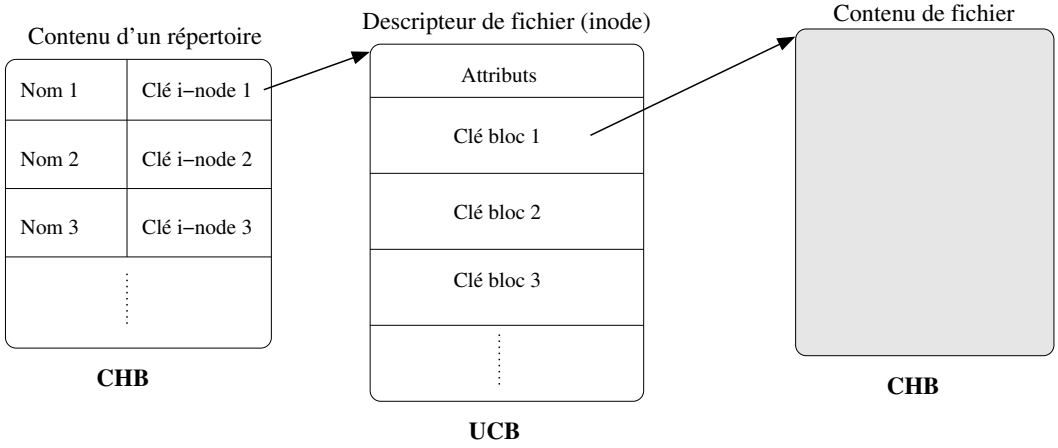

FIG. 4.3 – Le système de fichier Pastis.

Bien que ces systèmes pair-à-pair permettent le partage de fichiers modifiables à grande échelle, ils ont été conçus comme des systèmes de gestion de fichiers et proposent des modèles de cohérence très relâchés qui ne conviennent généralement pas pour le partage des données des applications de calcul scientifique.

4.3 Modèles et protocoles de cohérence dans les bases de données

La gestion des données dans les bases de données s'éloigne quelque peu des approches décrites jusqu'ici. Cette différence est principalement due au fait que les *systèmes de gestion de bases de données* (SGBD) prennent en compte le type des données ainsi que des dépendances interdonnée. On rencontre cependant les mêmes problématiques liées à la cohérence des données : présence de multiples copies pour améliorer la localité des accès, propagation des mises à jour, accès concurrents aux données, etc.

4.3.1 Particularité des données

Au sein des systèmes de bases de données, les données sont réparties dans des *tables* et chaque ligne d'une table forme un ensemble d'*attributs* appelé *tuple*. Un *schéma relationnel* décrit les tables ainsi que les relations entre les données. Il existe des dépendances intratuple, c'est-à-dire entre les attributs d'un même tuple, appelées *dépendances fonctionnelles* (*df*) et des dépendances entre les attributs de tables différentes appelées *dépendances d'inclusion* (*di*).

Définition 4.1 : ***dépendance fonctionnelle*** — Il y a une dépendance fonctionnelle lorsque la valeur d'un attribut (ou d'un groupe d'attributs) détermine de façon unique celle d'autres attributs.

Définition 4.2 : ***dépendance d'inclusion*** — Il y a une dépendance d'inclusion lorsque les valeurs d'un attribut d'une table doivent appartenir à l'ensemble des valeurs d'un attribut d'une autre table.

La gestion de la cohérence au sein des systèmes de bases de données intervient à deux niveaux : 1) entre les données (les attributs), ce qui est spécifique aux bases de données ; et 2) entre les copies des données répliquées.

Cohérence interdonnée. Le premier niveau, spécifique aux bases de données, vient de la gestion des dépendances interdonnée par les systèmes de gestion de bases de données. Les SGBD utilisant des schémas relationnels pour décrire les relations entre les données doivent prendre en compte les dépendances entre les différentes données. En conséquence, la mise à jour d'une donnée doit se faire de manière cohérente avec celles qui lui sont liées. Il peut donc être nécessaire de mettre à jour de manière atomique un ensemble de données. On parle alors de *transaction*.

Cohérence de données répliquées. De la même manière que pour les systèmes à mémoire virtuellement partagée ou les systèmes pair-à-pair, les systèmes de bases de données utilisent la réplication afin d'améliorer les performances des accès aux données. Elle favorise en effet la localité des accès d'une part, et la répartition de la charge d'autre part. Des mécanismes pour propager les mises à jour entre les multiples copies sont alors nécessaires.

Les deux sous-sections suivantes détaillent ces deux niveaux de cohérence.

4.3.2 Notion de transaction

La cohérence interdonnée est assurée par la notion de *transaction*. Nous reprenons la définition donnée dans [88].

Définition 4.3 : *transaction* — Une transaction est une suite d'opérations d'écritures ou de lectures sur des données qui se termine en étant soit *validée*, soit *abandonnée*.

Les transactions doivent présenter les propriétés suivantes : 1) *atomicité*, 2) *cohérence*, 3) *isolation*, et 4) *durabilité*.

Atomicité. Une transaction doit permettre de mettre à jour de manière atomique un ensemble de données dépendantes les unes des autres.

Cohérence. La base de donnée doit restée cohérente après une transaction.

Isolation. Une transaction n'a pas d'effet visible tant qu'elle n'a pas été validée.

Durabilité. Une transaction validée verra ses effets perdurer.

Ces propriétés sont généralement appelées "ACID" pour Atomicité, Cohérence, Isolation et Durabilité. Cependant, les SGBD peuvent relâcher la propriété d'isolation afin d'améliorer le parallélisme et donc les performances des accès (transactions optimistes). Dans ces cas-là, certains accès peuvent être effectués sur des données dont la valeur est obsolète, ou non encore validée. Ces données peuvent d'ailleurs ne jamais être validées et on parle alors de *divergence de transaction*. Autoriser les divergences des transactions en écriture peut s'avérer très dangereux, les dépendances interdonnée pouvant se retrouver non-satisfaites et la base de donnée serait alors dans un état *corrompu*. En revanche, certaines transactions de lecture peuvent se satisfaire de données "légèrement" obsolètes. Dans ce cas, il est possible d'améliorer fortement les performances [88] en relâchant les contraintes sur la fraîcheur des données lues.

4.3.3 Cohérence de données répliquées : divergence et réconciliation

Lors de l'utilisation des bases de données, les requêtes d'accès aux données peuvent être coûteuses : elles nécessitent souvent le parcours de tables entières. La réplication des données permet dans ce cas non seulement d'améliorer la localité des accès, mais également de répartir la charge engendrée par les requêtes entre plusieurs nœuds. Les mécanismes de réplication responsables de la propagation des mises à jour entre les différentes copies se déclinent en plusieurs catégories. On distingue la réplication *synchrone* et la réplication *asynchrone*. Cette dernière catégorie se décline en deux sous-catégories, la réplication *asynchrone optimiste* et la réplication *asynchrone pessimiste*. Ces différentes catégories de réplication sont décrites ci-dessous.

Réplication *synchrone*. Ces mécanismes sont similaires à ceux vus à la section 3.3.2, il s'agit en général de réplication *semi-active*. Toute modification se fait avec des propriétés d'atomicité, toutes les copies d'une même donnée sont isolées afin de valider chaque modification [35]. Les transactions ne sont donc validées que lorsque que toutes les copies sont cohérentes.

Réplication *asynchrone optimiste*. Ces mécanismes diffèrent la propagation des mises à jour. Des modifications concurrentes peuvent avoir lieu sur des copies différentes. Dans ces cas-là, il est nécessaire de *réconcilier* les copies, généralement en réordonnançant et ré-exécutant les transactions ou en abandonnant l'une (ou certaines) d'entre elles. Les réconciliations étant très coûteuses, ce type de réplication est généralement utilisé lorsqu'il est possible de faire l'hypothèse que les cas de conflits sont rares [2, 52].

Réplication *asynchrone pessimiste*. La réplication asynchrone pessimiste garantit la cohérence des copies en ordonnançant les transactions *a priori* afin d'éviter les conflits [87]. Ce type de réplication peut mener à des copies temporairement divergentes, le temps de la propagation des mises à jour, mais ne nécessite pas la mise en place de mécanismes de réconciliation : les différentes copies évoluent de la même manière et donc convergent ultimement.

Les systèmes de gestion de bases de données répliquées font face à une double problématique de cohérence de données. Ils doivent non seulement prendre en compte les différentes copies de chaque donnée, mais également maintenir la cohérence interne de la base de données. Nous proposons au chapitre 8 un mécanisme permettant des lectures relâchées à rapprocher du relâchement induit par les techniques de réplication asynchrone pessimiste.

4.4 Cohérence de données dans les grilles : vers une approche hiérarchique

Comme présenté dans les sections précédentes, la gestion de la cohérence de données a fait l'objet de recherches dans différents domaines. Dans chacun de ces domaines, des solutions spécifiques ont été proposées. Le tableau ci-dessous résume les caractéristiques des solutions décrites dans ce manuscrit.

Les systèmes à mémoire virtuellement partagée, conçus pour des applications scientifiques à haute performance et à petite échelle (en général quelques dizaines, et au maximum quelques centaines de nœuds), offrent la propriété de transparence et permettent la mise

	Systèmes à MVP	Systèmes P2P	Bases de données répliquées
Passage à l'échelle	Non	Très bon	Bon
Tolérance aux fautes	Non	Très bonne	Bonne
Modèles et protocoles de cohérence	Nombreux	Exceptionnels (lecture seule en général)	Spécifiques aux bases de données

en œuvre de nombreux modèles et protocoles de cohérence. En revanche, ils ne prennent généralement pas les fautes en compte, ou considèrent alors qu'elles sont exceptionnelles. Les systèmes pair-à-pair, quant à eux, présentent des mécanismes permettant un bon support de la volatilité et s'adaptent bien aux architectures à grande échelle, mais offrent peu de mécanismes pour la gestion de la cohérence des données modifiables. Enfin, des travaux de recherche dans le cadre des bases de données proposent des mécanismes variés de réplication ainsi que des solutions élégantes permettant de réconcilier des données divergentes. Cependant la gestion des données par les SGBD est très spécifique et ne s'adapte pas directement aux applications scientifiques que nous visons.

Dans le cadre des grilles de calcul, il semble important d'offrir : 1) les propriétés de passage à l'échelle ; 2) des modèles et protocoles de cohérence pour des applications scientifiques distribuées ; et également 3) un support pour la volatilité. Bien que dans le cadre des grilles de calcul la volatilité soit bien moins importante que dans le cadre des systèmes pair-à-pair, les déconnexions de nœuds ne peuvent plus être considérées comme des exceptions comme c'est le cas pour les systèmes à mémoire virtuellement partagée. Dans la partie suivante, nous décrivons notre contribution qui consiste à offrir une gestion de la cohérence tolérante aux fautes, en s'inspirant à la fois des mécanismes de gestion de la cohérence des données vus dans ce chapitre et des mécanismes de réplication vus dans le chapitre précédent.

Deuxième partie

Notre contribution : une approche hiérarchique conjointe pour la tolérance aux fautes et la cohérence des données

Chapitre 5

Étude de cas : vers un protocole de cohérence des données tolérant aux fautes pour la grille

Sommaire

Dans le chapitre 4, nous avons vu qu'il existait plusieurs approches pour la gestion du partage de données. Nous nous intéressons plus particulièrement au partage de données pour des applications de type calcul scientifique, comme les applications de couplage de code décrites à la section 2.2.

Les modèles de cohérence étudiés dans le cadre des systèmes à mémoire virtuellement partagée (MVP) ont été conçus puis optimisés pour ce type d'application. C'est en effet généralement pour des applications scientifiques (par exemple des simulations numériques distribuées) qui s'exécutent au sein des grappes de calculateurs que les systèmes à mémoire

virtuellement partagée ont été conçus. Pour cette raison, dans ce chapitre, nous prenons un exemple particulier de protocole de cohérence développé dans le cadre des MVP. Le protocole choisi implémente le modèle de cohérence à l'entrée et a été proposé par les auteurs du système à mémoire virtuellement partagée Midway [15].

Nous allons montrer que ce modèle convient bien aux applications visées. Cependant, les protocoles utilisés dans les systèmes MVP pour implémenter ce modèle de cohérence présentent des *limites* lorsque l'on essaie de les appliquer directement à une architecture de type grille, comme celle décrite à la section 2.1. Ces limites sont liées d'une part au *passage à l'échelle* des protocoles de cohérence et d'autre part à la *nature dynamique* des grilles de calcul.

Sur cet exemple, nous esquissons des idées de solutions pour faire face à ces limites. Ces solutions permettent d'*adapter le protocole de cohérence* aux architectures à *grande échelle et dynamiques*, que sont les grilles de calcul. Les chapitres suivants décrivent comment ces solutions ont été généralisées et intégrées lors de la conception du service de partage de données sur grille JUXMEM (pour *Juxtaposed Memory*).

5.1 Le partage de données

Dans un premier temps, nous décrivons notre vision du partage de données afin de mieux cerner les besoins. Les applications distribuées auxquelles nous nous intéressons sont les applications de type couplage de codes décrites à la section 2.2. Les processus de ces applications communiquent en partageant des données. Ce partage peut être géré soit de manière explicite, c'est-à-dire par l'application elle-même, soit par un système de partage de données extérieur à l'application, comme par exemple les systèmes MVP vus à la section 4.1 ou les systèmes pair-à-pair vus à la section 4.2. Nous n'étudions ici que le cas où le partage de données se fait via un système extérieur à l'application. En effet, nous avons montré dans la section 2.3 qu'il était préférable d'éviter la gestion explicite du partage de données par les applications, ceci devenant un facteur limitant lors du développement d'applications distribuées à grande échelle. Il nous faut commencer par définir ce qu'est une donnée partagée.

Donnée partagée. Une donnée partagée est un espace mémoire auquel peuvent accéder plusieurs processus d'une même application distribuée. Cela peut être une page mémoire, une variable, etc. Nous considérons dans ce manuscrit qu'une donnée partagée est une variable partagée, accédée par différents processus d'une application distribuée.

Accès à une donnée partagée. Lors de leur exécution, les processus ont besoin d'accéder aux variables partagées. Les *accès* à une variable partagée peuvent être de deux types : en *lecture* ou en *écriture*. Les lectures correspondent à une simple consultation de la valeur de la variable partagée. Les écritures quant à elles entraînent une *mise à jour* de la valeur de la variable.

Dans une application distribuée, les variables partagées permettent aux processus de prendre en compte les calculs effectués par les autres processus de l'application. La perception de ces variables par chacun des processus doit donc rester cohérente. Certaines questions se posent alors concernant la définition de la cohérence.

– Autorise-t-on les écritures concurrentes ?

- Quelle est la valeur finalement attribuée à la donnée en cas d'écritures concurrentes ?
- Autorise-t-on les lectures concurrentes à une écriture ?
- Quelle est la valeur retournée par une lecture d'une donnée lorsque qu'il y a une écriture concurrente ?
- Autorise-t-on les lectures concurrentes ?

Les applications qui partagent les données doivent savoir quelles garanties sont offertes en terme de cohérence. Par exemple : la valeur retournée par une lecture prend-elle toujours en compte toutes les écritures qui ont été effectuées auparavant ? Ou encore : un processus est-il seul à accéder à la donnée pendant une période de temps donnée ? Souvent, un processus doit pouvoir effectuer une lecture et une écriture sur une donnée partagée de manière atomique. Ce sera le cas notamment si un processus doit incrémenter la valeur d'une variable : $x := x + 1$. Pour rester cohérente, la valeur de la donnée ne doit pas être modifiée (c'est-à-dire qu'il ne doit pas y avoir d'écriture) entre la lecture (partie droite de l'affectation) et l'écriture (affectation).

Un contrat avec les applications. Les garanties de cohérence font l'objet d'un contrat entre les applications distribuées et le système qui gère le partage de données. Comme détaillé au chapitre 4, ce contrat est appelé *modèle de cohérence*. Dans ce chapitre, nous nous concentrons sur un exemple de protocole de cohérence implémentant le modèle de cohérence à l'entrée introduit par le système à mémoire virtuellement partagée Midway [15]. Ce modèle permet d'offrir des accès aux données performants, il convient particulièrement aux applications de calcul scientifique.

5.2 Un exemple de protocole non tolérant aux fautes

Les études de la cohérence des données dans les systèmes à mémoire virtuellement partagée ont montré que les modèles de cohérence relâchée pouvaient être implémentés par des protocoles efficaces. En revanche, les garanties de cohérence des données sont plus restreintes. Par exemple, les applications accédant aux données partagées doivent utiliser des opérations de synchronisation comme *acquire* ou *release*. L'opération *acquire* permet de s'assurer que les accès à la donnée qui suivront respecteront les garanties de cohérence. L'opération *release* permet quant à elle de garantir la propagation, éventuellement paresseuse, des mises à jours locales dans le système. Comme décrit dans la section 4.1.2, ceci est vrai pour des modèles comme la cohérence à la libération [51], la cohérence à l'entrée [15] et la cohérence de portée [61].

5.2.1 Le modèle de cohérence à l'entrée

Nous choisissons le *modèle de cohérence à l'entrée* pour cet exemple. Il présente la particularité d'associer explicitement un objet de synchronisation spécifique à chaque donnée. Cette particularité permet un gain de performance au niveau du protocole de cohérence. En effet, lorsqu'un processus entre en section critique[1] pour accéder à une donnée, par exemple pour

[1] Une section critique correspond à une portion de code manipulant des ressources communes ne devant être accédées que par un seul processus à la fois.

lire la valeur de la donnée avant de la mettre à jour, il doit le faire via l'objet de synchronisation associé à cette donnée. Ainsi, sur un nœud, le protocole de cohérence n'a à garantir que la cohérence des données accédées au sein de la section critique associée. Cela permet d'éviter des mises à jour non nécessaires : seuls les processus qui déclarent qu'ils vont accéder aux données seront concernés. De plus ne seront mises à jour que les données qui seront effectivement accédées.

Afin d'utiliser le modèle de cohérence à l'entrée, les programmeurs d'applications doivent respecter deux contraintes. Premièrement, chaque donnée partagée doit être associée à un objet de synchronisation spécifique. Deuxièmement, les accès *non-exclusifs* ou en *lecture seule* (c'est-à-dire *sans* mise à jour de la valeur de la donnée) doivent être distingués des *accès exclusifs* (c'est-à-dire *avec* mise à jour de la valeur de la donnée). La distinction entre les deux types d'accès est faite en utilisant des primitives différentes : *acquire_read* pour entrer dans une section de code où l'on effectue des accès non-exclusifs, et *acquire* pour entrer dans une section critique comprenant des accès de type exclusif. Ceci permet au protocole de cohérence d'autoriser des accès concurrents en lecture : plusieurs processus peuvent accéder en lecture, de manière concurrente, à une même donnée.

5.2.2 Un protocole basé sur une copie de référence

Comme point de départ pour cet exemple, nous choisissons un *protocole* implémentant le *modèle* de cohérence à l'entrée. Le protocole choisi est fondé sur la notion de *copie de référence* (*home based*) associée à chaque donnée partagée. Le nœud sur lequel l'unique copie de référence est placée (*home node*) est également responsable de la gestion de l'objet de synchronisation associé à la donnée.

Lorsqu'un processus accède à la donnée, il le déclare en verrouillant l'objet de synchronisation par l'intermédiaire d'une des primitives de synchronisation : *acquire* ou *acquire_read*. Lors de l'appel à l'une de ces primitives, la copie *locale*, située sur le même nœud que le processus, est mise à jour si nécessaire. De la même manière, lorsque le processus quitte la section au sein de laquelle il accédait à la donnée, il relâche l'objet de synchronisation par l'intermédiaire de la primitive *release*. Si la donnée a été modifiée par le processus durant la section critique, les mises à jour locales sont alors propagées à la copie de référence.

Il apparaît donc que de nombreuses communications sont nécessaires entre les nœuds sur lesquels s'exécutent les processus accédant à une donnée partagée et le nœud hébergeant la copie de référence associée à cette donnée. La figure 5.1 illustre les communications qui ont lieu entre le nœud hébergeant la copie de référence et celui sur lequel s'exécute un processus accédant à la donnée, appelé aussi *nœud client*.

5.2.3 Fonctionnement du protocole de cohérence

La figure 5.1 illustre le fonctionnement du protocole de cohérence lorsque les interactions sont limitées à un seul client. Il n'y a donc pas, dans cet exemple, de mise en attente au niveau du nœud hébergeant la copie de référence entre la réception d'une requête de synchronisation et l'envoi de l'autorisation d'accès : l'attente du processus client est limitée à la somme des temps de communications et du temps de traitement de la requête. Cependant, le protocole de cohérence gère aussi l'accès à une donnée *partagée* accédée par de multiples processus potentiellement de manière concurrente. Lorsque plusieurs processus accèdent

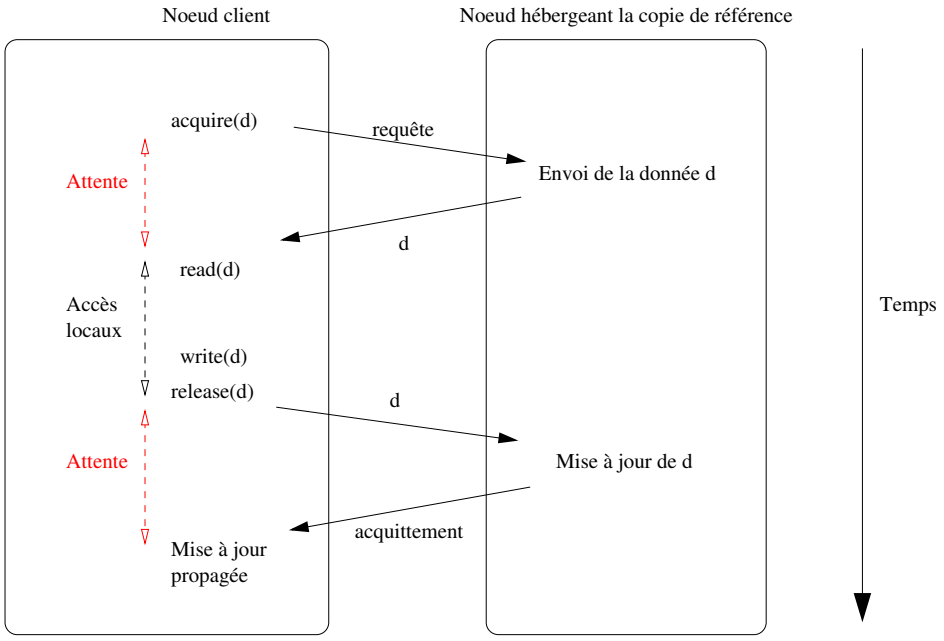

FIG. 5.1 – Communications entre un nœud client et un nœud hébergeant une copie de référence pour une donnée d.

de manière concurrente à la donnée le protocole met en attente certains processus afin de garantir la cohérence des accès.

Ce comportement est représenté par l'automate à états de la figure 5.2 qui décrit le fonctionnement simplifié du protocole de cohérence au niveau de la copie de référence. Cet automate est composé de trois états principaux : NO_LOCK, ON_LOCK et ON_LOCK_R. Ils correspondent respectivement aux cas où : 1) l'objet de synchronisation associé à la donnée est libre ; 2) un processus a acquis l'objet de synchronisation en mode exclusif ; et 3) un ou plusieurs processus ont acquis l'objet de synchronisation en mode non-exclusif. En plus de ces trois états principaux, le protocole de cohérence gère trois listes : 1) L (pour *Lock*) qui contient les identifiants des processus en attente pour un accès exclusif à la donnée ; 2) LR (pour *Lock Read*) qui contient les identifiants des processus en attente pour un accès non-exclusif ; et 3) OLR (pour *On Lock Read*) qui contient les identifiants des processus possédant l'objet de synchronisation en mode non-exclusif. On remarque notamment que le protocole ne permet un accès exclusif qu'à un seul processus client à un moment donné. En effet, quand il se trouve dans l'état ON_LOCK, il ne redistribue pas cet accès et il ne sort de cet état qu'à la réception d'un message de relâchement de l'objet de synchronisation.

L'automate à états représenté par la figure 5.3 décrit le fonctionnement du protocole de cohérence au niveau du nœud client. A ce niveau, le protocole ne gère pas de liste. En effet, pour une donnée, le client ne communique qu'avec la copie de référence associée. Il est intéressant de remarquer que certains états (ceux grisés sur la figure) sont bloquants pour l'application. Cela permet par exemple de garantir que le processus possède bien l'objet de synchronisation associé et que la copie locale de la donnée soit bien mise à jour avant de laisser l'application accéder à la donnée.

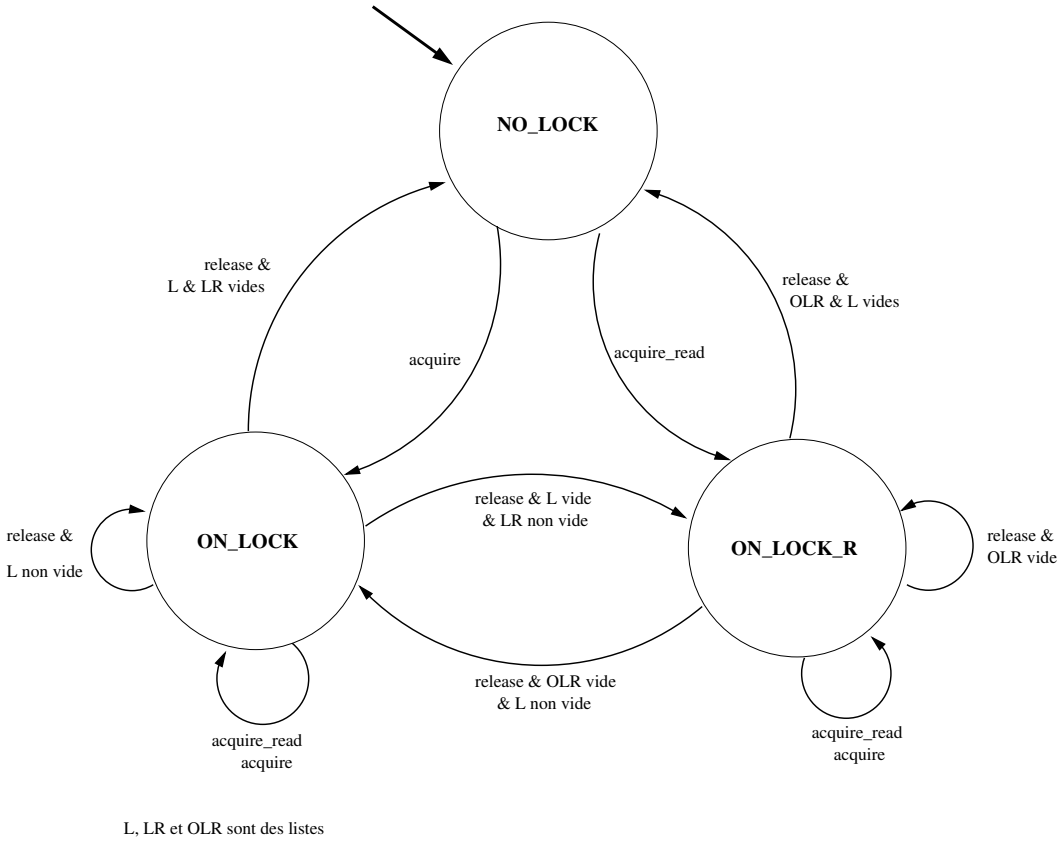

FIG. 5.2 – Automate à états représentant le protocole de cohérence au niveau de la copie de référence.

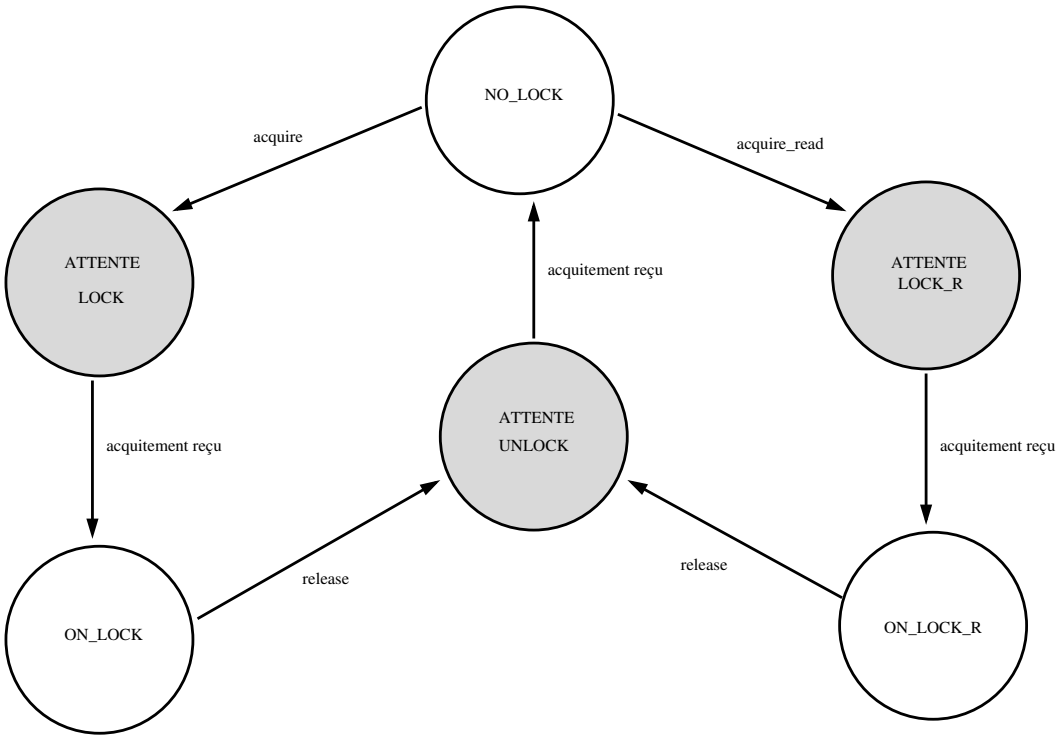

FIG. 5.3 – Automate à états représentant le protocole de cohérence au niveau du nœud client. Les états non grisés ne bloquent pas l'application alors que les états grisés correspondent à des primitives synchrones.

Le protocole décrit ici n'est pas adapté à une architecture de type grille : 1) il est centralisé autour du nœud hébergeant la copie de référence et ne prend pas en compte la topologie réseau des grilles ; 2) il ne prend pas en compte la possibilité d'apparition de fautes. Les sections suivantes détaillent ces limites et proposent des solutions pour y faire face.

5.3 Un protocole de cohérence hiérarchique

5.3.1 Limites d'un protocole "plat"

Comme l'illustrent les automates représentés par les figures 5.3 et 5.2, de multiples clients sont susceptibles d'accéder à une même donnée partagée. Les différents processus accédant à une même donnée partagée peuvent être situés dans des grappes différentes.

Si l'on se réfère à l'architecture de grille considérée comme une fédération de grappes présentée dans la section 2.1, les liens intergrappe présentent des latences réseau bien plus élevées que les liens intragrappe. Il en résulte que les processus clients qui s'exécutent sur un nœud situé dans une grappe différente de celle où se situe le nœud hébergeant la copie de référence bénéficieront d'accès bien moins performants que ceux s'exécutant sur des nœuds situés dans la même grappe.

Le point central qu'est la copie de référence constitue donc une limite à l'efficacité du protocole. La figure 5.4 illustre les communications entre deux processus clients, situés dans des grappes différentes, accédant à une donnée partagée. Le protocole ne fait aucune différence entre l'utilisation des liens intragrappe entre le client A et le nœud hébergeant la copie de référence et les liens intergrappe entre le client B et le nœud hébergeant la copie de référence. De plus, comme le montre l'automate 5.3, certains états du protocole de cohérence sont bloquants. Cela implique que la dégradation de performance concernera l'application distribuée toute entière, y compris les processus s'exécutant sur des nœuds situés dans la même grappe que la copie de référence, comme illustré par la figure 5.4.

5.3.2 Solution : un protocole hiérarchique

De manière à améliorer l'efficacité du protocole de cohérence, une approche consiste à *minimiser les communications entre les grappes*. Cette idée a été utilisée dans certains systèmes à mémoire virtuellement partagée et elle a conduit à la conception de *protocoles de cohérence hiérarchiques*. Dans *Clustered Lazy Release Consistency* (CLRC [6]), des caches locaux sont créés dans chaque grappe afin d'optimiser la localité des accès consécutifs à la donnée. Les auteurs de [5] proposent d'appliquer une approche hiérarchique à la gestion distribuée d'objets de synchronisation en réordonnançant les requêtes de verrouillage (par exemple *acquire*) : les requêtes provenant de la grappe courante sont servies avant les requêtes distantes.

En nous inspirant de ces deux contributions, nous construisons une version hiérarchique du protocole de cohérence décrit à la section 5.5. Cette version, illustrée par la figure 5.5, est inspirée par le protocole de synchronisation hiérarchique décrit dans [5]. L'idée est d'utiliser une hiérarchie à deux niveaux de copies de référence. Au sein de chaque grappe dans laquelle la donnée est accédée, une copie de référence locale est placée sur un nœud responsable des accès locaux, c'est-à-dire provenant d'un nœud de la même grappe. Une copie de référence globale est responsable des accès effectués par les copies de références locales.

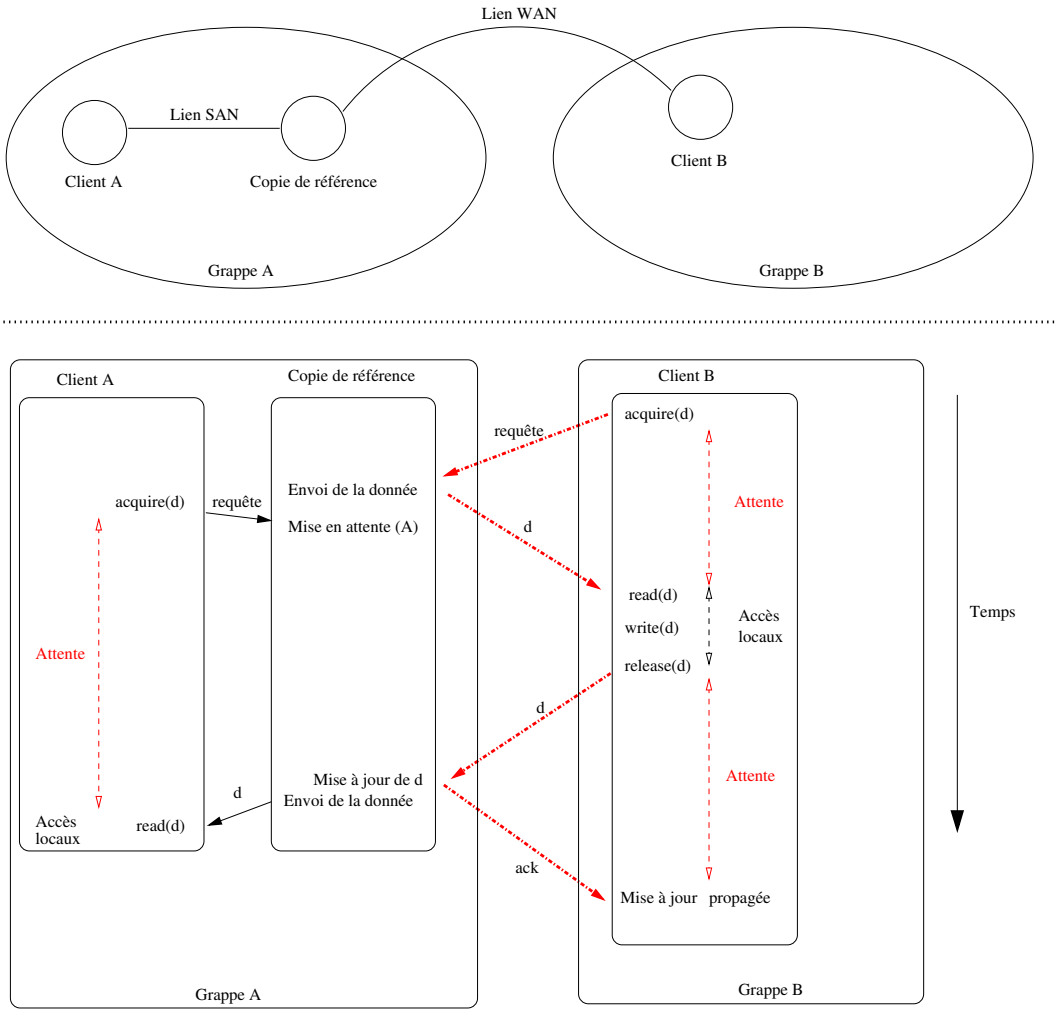

FIG. 5.4 – Fonctionnement du protocole de cohérence à l'entrée pour une donnée accédée dans des grappes différentes.

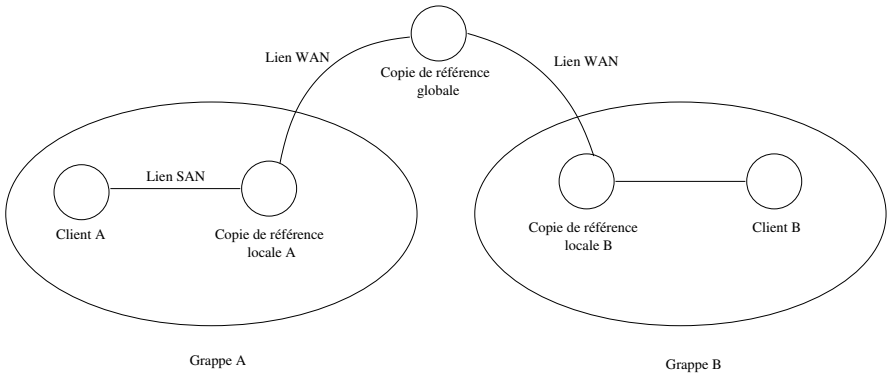

FIG. 5.5 – Version hiérarchique du protocole de cohérence.

Lorsqu'un client accède à la donnée, il doit d'abord acquérir l'objet de synchronisation correspondant auprès du nœud hébergeant la copie de référence locale. S'il possède l'objet de synchronisation, il peut autoriser l'accès. Si ce n'est pas le cas, une demande d'acquisition est envoyée au nœud hébergeant la copie de référence globale.

La copie de référence globale ne sert donc que les accès des copies de référence locales qui distribuent ensuite les accès au sein de leur grappe. Cependant, afin de minimiser les communications entre les grappes, à plus forte latence, les copies de références locales servent en priorité les demandes d'accès des clients de leur grappe par rapport aux requêtes distantes provenant des autres copies de référence locales et transmises via la copie de référence globale. Cette règle, a priori, peut aboutir à une situation de famine ; aussi une limite est-elle imposée sur le nombre maximal de redistributions locales de l'objet de synchronisation qui peuvent être effectuées par le nœud hébergeant une copie de référence locale. Cela permet d'assurer que les requêtes distantes seront ultimement satisfaites.

L'automate du protocole de cohérence au niveau des copies de référence locales est une combinaison des deux automates de la version non hiérarchique du protocole de cohérence. Les copies de référence locales jouent en effet le rôle de copie de référence vis-à-vis des clients locaux, et le rôle de client vis-à-vis de la copie de référence globale. L'automate de la copie de référence globale est identique à celui de la copie de référence unique présenté lors de la description de la version non hiérarchique du protocole de cohérence (figure 5.2). Cela est dû au fait que la copie de référence globale de la version hiérarchique joue, vis-à-vis des copies de référence locales, le même rôle que la copie de référence du protocole en version non hiérarchique jouait vis-à-vis des clients.

De même, au niveau du client, l'automate est identique à celui de la version non hiérarchique (voir figure 5.3). En effet, du point de vue du client, la hiérarchisation du protocole est transparente : tout se passe comme si la copie de référence locale, à laquelle il accède, était la seule copie de référence du système. Le nouveau rôle créé par la hiérarchisation du protocole de cohérence est celui de *copie de référence locale*, qui, en jouant le rôle *d'intermédiaire*, permet de limiter l'utilisation des liens réseau à forte latence entre les grappes et ainsi d'améliorer les performances globales des accès à la donnée partagée. La copie de référence globale peut être localisée au sein de l'une des grappes contenant des nœuds accédant à la donnée, ou dans une grappe tierce. La question de la localisation de la copie globale est discutée plus en

détails dans le chapitre 7.

Cette version du protocole de cohérence est donc mieux adaptée aux grilles de calcul. Cependant, nous avons vu à la section 2.1 que la possibilité d'occurrence de fautes dans les grilles de calcul n'est pas négligeable. Dans la section suivante, nous montrons comment il est possible de rendre le protocole de cohérence *tolérant aux fautes*.

5.4 Un protocole de cohérence tolérant aux fautes

5.4.1 Nécessité de tolérer les fautes

Le protocole de cohérence à l'entrée que nous avons décrit à la section 5.2 a été mis au point pour des systèmes à mémoire virtuellement partagée. Ces systèmes s'exécutant généralement sur des grappes de calculateurs dédiés de taille restreinte (quelques dizaines, voire quelques centaines de nœuds), les fautes y sont considérées comme *exceptionnelles*. De ce fait, lors de la conception de ce protocole, le nœud hébergeant la copie de référence peut être considéré comme *stable* (c'est-à-dire *correct* ou *non fautif*). Or nous visons l'échelle de la grille et la gestion de nombreuses données partagées, ce qui implique que de nombreux nœuds (des milliers) soient impliqués dans la gestion des données. À cette échelle, comme expliqué dans la section 2.1, les fautes ne peuvent plus être considérées comme *exceptionnelles* : elle sont partie intégrante des *propriétés de la grille*. Il n'est donc plus possible de considérer qu'un nœud hébergeant une copie de référence pour une donnée soit stable. En effet, étant donné le nombre de données que nous allons gérer au sein de la grille, le nombre de copies de référence sera suffisamment important pour que l'apparition d'une faute sur un des nœuds jouant ce rôle pour une donnée particulière soit un événement non négligeable.

Nous reprenons ici le protocole non hiérarchique et non tolérant aux fautes décrit à la section 5.2. Le rôle de copie de référence peut être considéré comme *critique* car chaque donnée partagée se voit associer *une seule et unique* copie de référence. Si le nœud hébergeant cette copie de référence est fautif, alors la donnée associée est perdue et les clients ne peuvent plus y accéder. Assurer la persistance de la copie de référence dans le système permet d'assurer la persistance de la donnée en cas de fautes.

Modèle de faute. Nous nous intéressons aux deux types de faute qui doivent être pris en compte dans les environnements de type grille : 1) les *pannes franches* (ou *crashs*)[2] ; et 2) les pertes de messages (canaux de communication non fiables). Ces types de faute sont représentatifs dans le cas des grilles de calcul comme expliqué dans la section 3.1.1.

Modèle de temps. Nous considérons que des bornes sur les temps de communications et les temps de calculs existent, mais ne sont pas connues. Nos algorithmes sont donc conçus pour un environnement partiellement asynchrone et utilisent des informations provenant d'un service de détection de fautes. Un tel service fournit sur chaque nœud une liste de nœuds suspectés d'être fautifs. Cette hypothèse d'asynchronisme est classique dans les systèmes distribués tolérants aux fautes. Elle est également réaliste dans le cas des grilles de calcul (voir section 2.1).

[2]Les nœuds qui rejoignent le système après une faute sont traités comme de nouveaux nœuds

5.4.2 Utilisation de techniques de réplication

Afin qu'une copie de référence puisse rester disponible en présence de pannes franches de nœuds, nous nous appuyons sur les solutions basées sur la *réplication* présentées à la section 3.3. Répliquer la copie de référence associée à une donnée permet d'assurer sa persistance en présence de fautes. Le nombre de fautes (de type *panne franche*) tolérées dépend de : 1) la technique de réplication employée ; et 2) du degré de réplication (le nombre de copies).

Afin de tolérer les fautes nous proposons donc de remplacer la copie de référence unique par un *groupe de copies de références*. Nous allons donc répliquer, sur différents nœuds du groupe, non seulement la donnée mais également le protocole de cohérence (l'automate à états). Afin que les automates et la valeur de la donnée évoluent de la même manière sur chacun des automates, les algorithmes utilisés doivent assurer la *diffusion atomique* des messages au sein du groupe. Ces mécanismes sont coûteux mais permettent d'offrir de bonnes garanties de tolérance aux fautes. L'architecture que nous proposons au chapitre suivant permet de choisir le type de réplication et donc le compromis coût/garanties de la tolérance aux fautes.

Pour réaliser la réplication de la copie de référence décrite ci-dessus, nous utilisons des concepts étudiés dans le cadre des systèmes distribués tolérants aux fautes.

Protocole de composition de groupe. Un protocole de *composition de groupe* (*group membership* en anglais) [37] permet de gérer un ensemble (un groupe) de nœuds ayant un intérêt commun. Chaque nœud appartenant à un groupe maintient à jour la liste de membres de ce groupe. La composition de ces listes évolue quand de nouveaux nœuds joignent ou quittent le groupe, lors d'une panne franche par exemple. Les protocoles de composition de groupe ont pour objectif de maintenir la cohérence de ces différentes listes en les synchronisant. Entre deux synchronisations, les mêmes ensembles de messages doivent être pris en compte par chacun des membres du groupe. Dans notre cas, nous utilisons ce type de protocole pour gérer les groupes de nœuds hébergeant les copies de référence des données.

Diffusion atomique. Dans la version tolérante aux fautes du protocole de cohérence, la copie de référence associée à une donnée est représentée par un groupe de nœuds. Étant donné que des nœuds peuvent être fautifs, nous utilisons un mécanisme de réplication pessimiste afin d'assurer qu'une copie à jour reste disponible dans le système. Chaque mise à jour entraîne donc une mise à jour de *tous* les membres du groupe. Afin de réaliser cela, tous les messages envoyés à un groupe doivent être pris en compte dans le même ordre par chacun des membres du groupe. Les membres du groupe s'accordent sur l'ordre de distribution des messages en utilisant un protocole "classique" de consensus, comme celui décrit dans [32].

Protocole de consensus. Un protocole de consensus permet à un groupe de nœuds potentiellement fautifs de s'accorder sur une valeur commune : chaque nœud propose une valeur et le protocole de consensus assure que : 1) ultimement, tous les nœuds non fautifs déciderons d'une valeur de manière déterministe ; 2) la valeur décidée est l'une des valeurs proposée ; et 3) la valeur décidée est la même pour tous les nœuds non fautifs. Dans notre cas, la décision porte sur l'ordre dans lequel les messages doivent être pris en compte. Le problème du consensus dans les systèmes asynchrones peut être

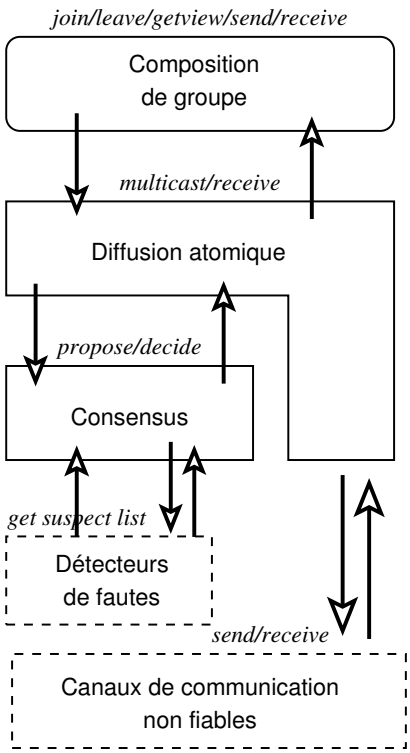

FIG. 5.6 – Architecture en couches pour l'implémentation du protocole de composition et de communication de groupe.

résolu grâce à des détecteurs de fautes non fiables [32], comme ceux présentés dans la section 3.2.

Pour réaliser une version tolérante aux fautes du protocole de cohérence présenté à la section 5.2, nous combinons des blocs de base implémentant les concepts décrits ci-dessus. La combinaison en couches illustrée par la figure 5.6 est inspirée par [82]. La mise en œuvre de l'architecture illustrée par la figure 5.6 pour répliquer la copie de référence associée à une donnée permet de supporter $\lfloor \frac{n-1}{2} \rfloor$ fautes simultanées, où n est le nombre de nœuds sur lesquels la copie de référence est répliquée. Cette limite est imposée par notre mise en œuvre du protocole de consensus proposé par Chandra et Toueg [32].

Notons que le protocole de cohérence reste inchangé aussi bien pour les clients que pour la copie de référence. Les clients utilisent une couche de communication de groupe pour communiquer avec la copie de référence répliquée, de la même manière qu'ils le faisaient lorsque cette dernière n'était hébergée que par un seul nœud. Le protocole de communication de groupe permet à chaque nœud hébergeant un réplicat de la copie de référence de s'abstraire de la réplication : chacun d'entre eux s'exécute comme s'il hébergeait la seule copie de référence. Les automates du protocole de cohérence sont donc les mêmes que dans la version non tolérante aux fautes présentée à la section 5.2 (figures 5.2 et 5.3).

5.5 Vers un protocole hiérarchique tolérant aux fautes

La section 5.3 a montré comment hiérarchiser le protocole de cohérence présenté à la section 5.2 afin que celui-ci prenne en compte la topologie réseau des grilles de calcul pour améliorer les performances des accès aux données partagées. Ensuite, la section 5.4 explique comment, en utilisant des concepts connus des systèmes distribués tolérants aux fautes, il est possible de rendre ce protocole tolérant aux fautes en répliquant les copies de référence.

L'étape finale est de coupler les deux solutions présentées dans les deux précédentes sections afin d'obtenir un protocole de cohérence à la fois hiérarchique et tolérant aux fautes. Cela permet de s'abstraire des limites du protocole de cohérence d'origine : 1) les problèmes liés au passage à l'échelle, et 2) le manque de support pour la tolérance aux fautes.

Une première idée pour combiner ces deux solutions consiste à utiliser les mécanismes de réplication décrits à la section 5.4 afin de rendre *tolérante aux fautes* la copie de référence globale du protocole de cohérence *hiérarchique* décrit à la section 5.3. Cependant, la faute d'un nœud hébergeant une copie de référence locale (susceptible de posséder l'objet de synchronisation et la dernière version de la donnée) doit également être tolérée. Il convient donc d'utiliser les mécanismes de réplication pour chacune des copies de référence (les copies locales et la copie globale).

Les mécanismes de tolérance aux fautes (comme la réplication) sont coûteux. Afin de limiter ce coût, il est important de limiter le nombre de nœuds que ces mécanismes font intervenir et de prendre en compte la nature des liens réseau utilisés (intergrappe ou intragrappe). Aussi, le couplage entre le protocole de cohérence et les mécanismes de tolérance aux fautes est un problème non trivial. Cette difficulté est encore augmentée si l'on souhaite concevoir une architecture permettant la mise en place de plusieurs protocoles de cohérence et de plusieurs mécanismes de tolérance aux fautes.

Le chapitre suivant décrit notre architecture logicielle permettant une gestion *conjointe* de la cohérence des données et de la tolérance aux fautes.

Chapitre 6

Une approche conjointe

Sommaire

Dans les chapitres 3 et 4, nous avons vu que plusieurs approches ont été proposées *séparément*, aussi bien pour la gestion de la tolérance aux fautes que pour la gestion de la cohérence des données. Dans le chapitre 5, nous avons montré sur un exemple des solutions pour adapter certaines de ces approches à la grille.

Dans le cadre d'un service de partage de données pour les grilles de calcul, il est nécessaire d'offrir un support pour la tolérance aux fautes *et* pour la gestion de la cohérence des données.

Ce chapitre décrit une architecture dont le but est de permettre une gestion *conjointe* de la tolérance aux fautes et de la cohérence des données au sein d'un service de partage de données pour la grille. Avant de détailler notre solution, nous décrivons le cadre dans lequel s'inscrit notre contribution : le service de partage de données JuxMem. Notre approche pour faire face aux problèmes liés au passage à l'échelle sera abordée dans le chapitre suivant.

6.1 Cadre : le service de partage de données JUXMEM

Comme nous l'avons vu à la section 2.2, les applications scientifiques utilisent des quantités de données distribuées de plus en plus grandes. La gestion de la localisation, du transfert et de la cohérence de ces données est devenue un facteur limitant lors de la conception de telles applications. Or il n'existe actuellement aucun système permettant un partage de données modifiables offrant un accès transparent et des garanties de cohérence pour les grilles de calculs. Ces besoins ainsi que les lacunes des mécanismes de partage de données pour les grilles existants sont détaillés dans la section 2.3. Une nouvelle approche consiste à externaliser la gestion des données partagées par les applications s'exécutant sur la grille. Le concept de service de partage de données pour la grille a été introduit par [4], la section suivante en donne notre vision.

6.1.1 Notion de service de partage de données pour la grille

Au vu des problèmes soulevés par la section 2.3, il apparaît que les propriétés recherchées pour le partage de données pour les grilles de calcul sont : 1) la persistance des données, 2) la cohérence des données, et 3) la transparence d'accès aux données.

Persistance des données. Les données partagées entre différentes entités doivent être persistantes. En effet, dans un système distribué, les opérations peuvent être asynchrones. Cela implique qu'une donnée utilisée par une entité d'une application distribuée pourra être utilisée à nouveau, *plus tard*, par une autre entité. La donnée, ainsi que les éventuelles modifications qui lui ont été apportées, doit rester disponible au cours du temps, jusqu'à ce quelle soit explicitement supprimée. Or, dans un environnement de type grille, les occurrences de fautes sont courantes. Le service de partage de données doit donc mettre en place des mécanismes de *tolérance aux fautes* afin d'assurer que les données ne seront pas perdues en cas de fautes.

Considérons l'exemple de DIET (pour *Distributed Interactive Engineering Toolbox* [28]) qui est une plate-forme de calcul développée à l'ENS Lyon au sein du projet de recherche GRAAL. Une donnée (une matrice par exemple) utilisée ou produite par un calcul peut être utilisée à nouveau par des calculs qui s'exécuteront plus tard. Entre temps, la donnée doit persister (rester disponible) dans le système afin de permettre aux calculs ultérieurs utilisant cette donnée de s'exécuter correctement. Si le service de partage de données n'offre pas cette propriété, c'est aux clients que revient le rôle de conserver les données, de se les transmettre et donc de maintenir la cohérence entre les différentes copies ainsi créées.

Cohérence de données. Une donnée *partagée* va, par définition, être accédée par de multiples entités d'une application distribuée et ce à la fois en lecture et en écriture (les écritures peuvent notamment être concurrentes). Un service de partage de données pour la grille doit prendre en compte cette possibilité et mettre en œuvre des protocoles de cohérence implémentant des modèles de cohérence, à l'image de ce qui a été fait dans les systèmes à mémoire virtuellement partagée (voir section 4.1). Cela signifie que le service doit assurer la cohérence des différentes copies de chaque donnée partagée par les clients (ordonner et propager les mises à jour, mettre en attente les processus lorsque cela est nécessaire, etc.).

Accès transparent. Un des facteurs limitant dans le développement d'applications scientifiques distribuées à l'échelle de la grille est la gestion explicite de la localisation et du transfert des données (et par conséquence, de leur cohérence). À l'image de ce qui existe dans les systèmes à mémoire virtuellement partagée et dans les systèmes pair-à-pair (P2P pour l'anglais *peer-to-peer*), un service de partage de donnée pour la grille se doit d'offrir un accès transparent aux données, par exemple en associant à chaque donnée un identifiant global unique. Cet identifiant peut être utilisé pour accéder à la donnée par chacune des entités de l'application sans avoir à connaître la localisation de la donnée, ni le mode de transfert utilisé.

À la frontière entre deux mondes. L'architecture physique visée par notre service de partage de données est celle des grilles de calcul et plus particulièrement celle des fédérations de grappes. Nous visons donc une échelle de l'ordre de quelques milliers de nœuds regroupés en grappes interconnectées. Il est intéressant de noter que cette architecture se situe à la frontière entre deux mondes : celui des systèmes à mémoire virtuellement partagée (MVP) d'une part et celui des systèmes pair-à-pair d'autre part. Ceci vaut en terme d'échelle, de volatilité, de degré de confiance et d'homogénéité, comme résumé sur le tableau 6.1. En effet, l'échelle que nous visons se situe entre les échelles généralement visées par ces deux mondes. De même le degré de volatilité, considéré comme nul dans les MVP et comme très élevé dans les systèmes pair-à-pair, est dans notre cas intermédiaire. Les nœuds sur lesquels s'exécute notre service font partie de grappes de calculateurs dédiés et sont moins volatiles que ceux considérés dans les systèmes pair-à-pair. En revanche, l'échelle visée implique tout de même un certain degré de volatilité, comme expliqué dans la section 3.1. Le degré de confiance (et par conséquence le niveau de contrôle des ressources) se situe également à mi-chemin : une fédération de grappe correspond généralement à la mise en commun de ressources entre différentes institutions qui se font confiance.

Les principales différences avec les systèmes à mémoire virtuellement partagée sont : 1) la présence de fautes, et 2) la structure hiérarchique du réseau. Ceci implique qu'il n'est plus possible de considérer des nœuds comme étant stables ni de supposer l'existence d'une connaissance globale de la plate-forme.

La principale différence avec les systèmes pair-à-pair classiques de partage de fichiers en lecture seule est que les données gérées par notre service de partage sont *modifiables*. Ce dernier point induit une grande complexité. En effet, il ne s'agit plus seulement de stocker de multiples copies en lecture seule puis de pouvoir en localiser au moins une. Il devient nécessaire de gérer la cohérence de ces données, c'est-à-dire de *toutes* les copies de *chaque* donnée.

Nous ne cherchons pas à concevoir un système pair-à-pair pour la grille, ni un système à MVP passant à l'échelle. Notre service de partage de données se situe au niveau intermédiaire. Il s'agit d'un service de grille s'inspirant :

1. des MVP pour leurs modèles et protocoles de cohérence de données ;
2. des systèmes pair-à-pair pour leurs propriétés de passage à l'échelle, notamment en terme de support de la volatilité et de localisation des ressources.

Dans la conception de notre service de partage de données pour la grille, nous avons exploité l'aspect hiérarchique des fédérations de grappes afin de faire le lien entre les deux mondes. Nous nous sommes également appuyés sur des résultats théoriques provenant de

	MVP	*Service de données pour la grille*	P2P
Échelle (nombre de nœuds)	10^1–10^2	*10^3*	10^4–10^6
Degré de volatilité	Nul	*Moyen*	Fort
Contrôle des ressources et Degré de confiance	Fort	*Moyen*	Nul
Homogénéité des ressources	Homogènes (grappes)	*Hiérarchique (fédérations de grappes)*	Hétérogènes (Internet)
Type de données gérées	Modifiables	*Modifiables*	Non modifiables
Applications typiques	Calcul numérique	*Calcul numérique et stockage de données*	Partage et stockage de fichiers

TAB. 6.1 – Caractéristiques d'un service de partage de données inspiré des MVP et du P2P.

la recherche sur les systèmes distribués tolérants aux fautes, notamment pour la conception de mécanismes de réplications.

6.1.2 Architecture générale

L'architecture générale de notre service de partage de données pour la grille JUXMEM a été conçue en collaboration avec Mathieu Jan [64], dans le cadre de nos travaux de doctorat au sein de l'équipe PARIS de l'IRISA.

Une architecture hiérarchique. JUXMEM utilise la notion de groupes de pairs afin de prendre en compte l'architecture physique sous-jacente. Ceci est illustré par la figure 6.1. L'architecture visée étant de type *fédération de grappes*, les pairs localisés dans une même grappe sont regroupés dans un groupe *cluster*. Il y a par conséquent autant de groupes *cluster* que de grappes utilisées au sein de la fédération. L'ensemble des groupes *cluster* forme un groupe de plus haut niveau appelé le groupe JUXMEM. Ce groupe rassemble tous les pairs participant au service. JUXMEM présente donc une architecture hiérarchique à deux niveaux : le niveau global et le niveau local (aussi appelé niveau *grappe*).

Rôles des pairs JUXMEM. Les pairs, entités constituantes des groupes, peuvent avoir à jouer différents rôles. Nous en distinguons trois : 1) les pairs *gestionnaires* qui jouent notamment le rôle de pairs de rendez-vous du réseau logique pair-à-pair basé sur JXTA [115] ; 2) les pairs *fournisseurs* sur lesquels les copies des données gérées par le service vont effectivement être stockées ; et 3) les pairs *clients* qui ont la capacité d'accéder aux données. C'est exclusivement avec ces derniers que les applications utilisant JUXMEM interagissent. Étant

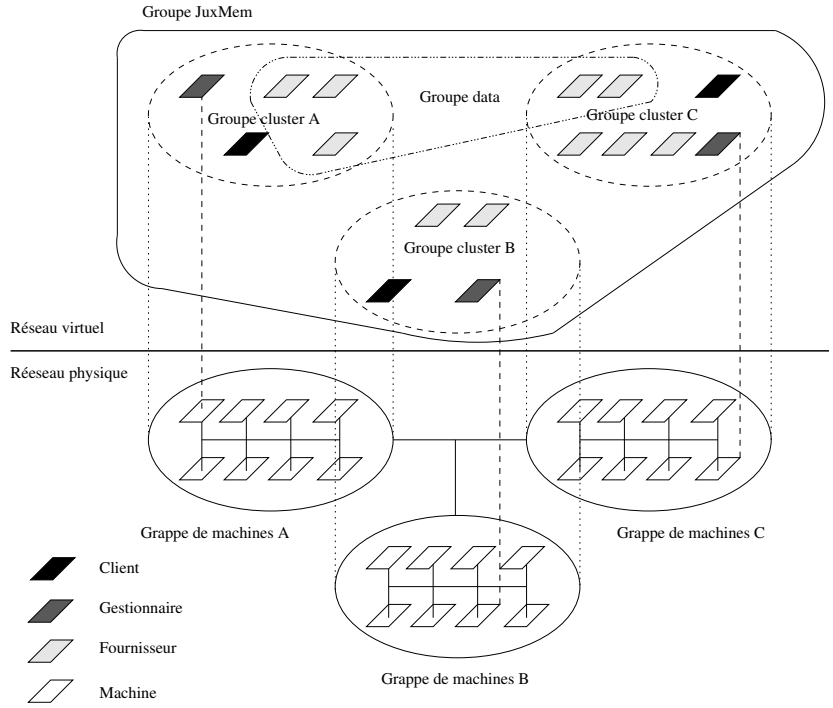

FIG. 6.1 – JUXMEM : une proposition d'architecture pour un service de partage de données pour la grille.

dans un monde pair-à-pair, un pair peut jouer un ou plusieurs rôles simultanément. Par exemple, un pair peut jouer à la fois le rôle de gestionnaire et de fournisseur de données. Par la suite, à des fins de clarté, nous ne considérerons que les cas où chaque pair ne joue qu'un seul rôle ; de plus nous considérons que chaque nœud physique héberge un seul pair[1].

Les groupes *data*. Nous allons nous intéresser plus particulièrement aux pairs de type fournisseur qui seront les membres des groupes *data*. Un groupe *data* est associé à chaque donnée gérée par JUXMEM. Il est constitué de l'ensemble des pairs fournisseurs possédant une copie de cette même donnée. Comme l'illustre la figure 6.1, un groupe *data* peut s'étendre sur plusieurs groupes *cluster*. Ces groupes ne sont pas des groupes au même sens que les groupes *cluster* ou le groupe JUXMEM. En effet, les garanties requises pour la gestion de la cohérence des copies d'une même donnée nécessite une gestion spécifique des groupes *data* pour assurer justement la gestion conjointe de la tolérance aux fautes et de la cohérence des données au sein de ces groupes. La description détaillée de la gestion de ces groupes constitue donc le coeur de ce manuscrit.

Principe d'utilisation de JUXMEM. Les groupes *data* jouent un rôle important dans l'utilisation de JUXMEM. Les applications utilisant JUXMEM sont liées à un *client* JUXMEM : chaque processus applicatif accédant à des données partagées interagit avec un pair client JUXMEM s'exécutant sur le même nœud. Lorsqu'un espace de stockage est alloué dans JUXMEM, un

[1]Dans la réalité un nœud peut héberger plusieurs pairs jouant chacun plusieurs rôles.

ensemble de pairs fournisseurs est sélectionné afin de former un nouveau groupe *data*. Par la suite, les accès (en lecture ou écriture) à la donnée présente dans l'espace de stockage impliquent des interactions entre le pair *client* et le groupe de pairs *fournisseurs* (c'est-à-dire le groupe *data*). La gestion des groupes *data* ainsi que les interactions entre ces groupes et les pairs clients sont détaillées par la suite.

6.1.3 Le noyau JuxMem

Dans la section suivante, nous allons décrire notre architecture logicielle pour la gestion des groupe *data*. Celle-ci s'appuie sur le noyau du système JUXMEM appelé *juk*. La réalisation de *juk* a été effectuée dans le cadre du doctorat de Mathieu Jan [64]. C'est le noyau qui gère notamment le groupe JUXMEM et les groupes *cluster*. Il est également responsable des interactions avec les applications.

Ce noyau sert de base aux mécanismes de tolérance aux fautes et de cohérence de données. Il offre aux couches supérieures la notion de pairs identifiés par des identifiants *uniques*[2], des mécanismes de communication, de stockage et de publication/recherche. Ces différents mécanismes s'appuient sur JXTA [115]. Au niveau des couches supérieures décrites dans ce manuscrit, ce noyau permet de s'abstraire complètement de JXTA. Son interface est détaillée ci-dessous.

Communication. Le noyau permet d'envoyer des messages d'un pair à un autre selon le modèle point-à-point. Afin de pouvoir recevoir des messages, les algorithmes doivent enregistrer auprès du noyau une fonction et une *étiquette* associée (simple chaîne de caractères). La fonction sera appelée lors de l'arrivée d'un message étiqueté avec l'étiquette associée. Pour envoyer un message, il faut fournir l'identifiant du pair destinataire ainsi que l'étiquette (send(identifiant, message, etiquette)). La couche de communication gère la localisation du pair destinataire. Il est important de noter que cette couche est asynchrone et non-fiable : la méthode d'envoi de message rendra la main (non bloquante) sans garantir que le message envoyé sera reçu par le pair destinataire. Nous faisons cependant l'hypothèse qu'elle est équitable (voir section 3.1) : un message ré-émis suffisamment souvent finira par être reçu.

Stockage. C'est également le noyau qui gère le stockage physique sur le nœud (en mémoire ou sur disque) en offrant des primitives comme malloc pour allouer de l'espace sur le nœud, data_to_message pour placer la copie de la donnée stockée sur le nœud dans un message, data_from_message pour stocker sur le nœud une donnée reçue dans le corps d'un message. Les algorithmes de réplication ainsi que ceux de cohérence n'auront donc pas à se préoccuper du stockage effectif de la donnée.

Publication/recherche. Les fonctionnalités de publication/recherche offertes permettent de publier des annonces et d'en rechercher. La publication/recherche s'effectue au sein d'un groupe. Il est ainsi possible d'effectuer une publication ou une recherche soit au niveau d'un groupe *cluster* (locale), soit au niveau du groupe *juxmem* (globale). Ceci peut être par exemple exploité pour retrouver les copies d'une donnée. C'est également

[2] L'unicité des identifiants n'est pas totalement garantie. Une fonction de hachage génère des identifiants qui ont une très grande probabilité d'être uniques. Par la suite, nous faisons l'hypothèse habituelle que la propriété d'unicité des identifiants est vérifiée.

cette possibilité de recherche *locale* ou *globale* qui nous permet de prendre en compte la hiérarchie physique sous-jacente.

Recherche de pairs fournisseurs. Le noyau offre également un mécanisme de recherche particulier qui est utilisé lors de l'allocation. Chaque pair de type *fournisseur*, lors de son initialisation, publie la quantité de mémoire disponible qu'il offre. Il est par la suite possible de rechercher des pairs JUXMEM *fournisseurs* pouvant allouer une quantité de mémoire donnée. Cette recherche peut être paramétrée plus finement : il est en effet possible de stipuler que l'on souhaite trouver des pairs fournisseurs présents dans le groupe *cluster* dans lequel se trouve le pair (généralement de type *client*) ayant lancé la requête d'allocation, ou, au contraire, situés dans d'autres groupes *cluster*. Il est également possible de rechercher des fournisseurs situés dans un nombre donné de groupes *cluster* distincts. Le noyau n'effectue pas les requêtes d'allocation proprement dites, il cherche des identifiants de pairs fournisseurs satisfaisant la requête et retourne la liste des identifiants trouvés. C'est le protocole de cohérence côté client qui, lors de son initialisation, envoie les requêtes d'allocation aux pairs *fournisseurs* retournés afin de créer un nouveau groupe *data*.

Interface avec les applications. Le noyau JUXMEM est également responsable des interactions avec les applications utilisant le service de partage de données. C'est lui qui appelle les primitives des protocoles de cohérence lorsque l'application accède à la donnée.

La figure 6.2 illustre l'architecture logicielle de JUXMEM. Notre contribution, décrite en détail dans les sections suivantes, se situe entre la partie du noyau responsable des interactions avec les applications et celle offrant les primitives de base décrites ci-dessus.

6.2 Proposition : une architecture en couches

Au-dessus du noyau JUXMEM présenté ci-dessus, nous proposons une architecture en couches dont le rôle est d'assurer la persistance des données en présence de fautes ainsi que leur cohérence. Cette architecture est présente au niveau des pairs de type *fournisseur* et au niveau des pairs de type *client*. En effet, le rôle de l'architecture logicielle présentée ci-dessous est la gestion des groupes *data* et les interactions entre les groupes *data* et les pairs *client* JUXMEM. L'ensemble des couches est conçu pour gérer une donnée partagée, il peut donc être instancié sur un pair *fournisseur* autant de fois que le nombre de données différentes stockées par ce dernier (le nombre de groupes *data* auxquels il appartient). De même, sur les pairs de type *client*, ces couches sont instanciées autant de fois que le nombre de données partagées auxquelles accède le client.

6.2.1 Un double besoin de réplication

Notre architecture doit gérer à la fois la tolérance aux fautes et la cohérence des données. Pour la tolérance aux fautes, la réplication permet d'améliorer la *disponibilité* des données. Dans le cas des protocoles de cohérence, la réplication est employée afin d'améliorer la *localité* des données.

FIG. 6.2 – Le noyau du système JUXMEM et notre contribution.

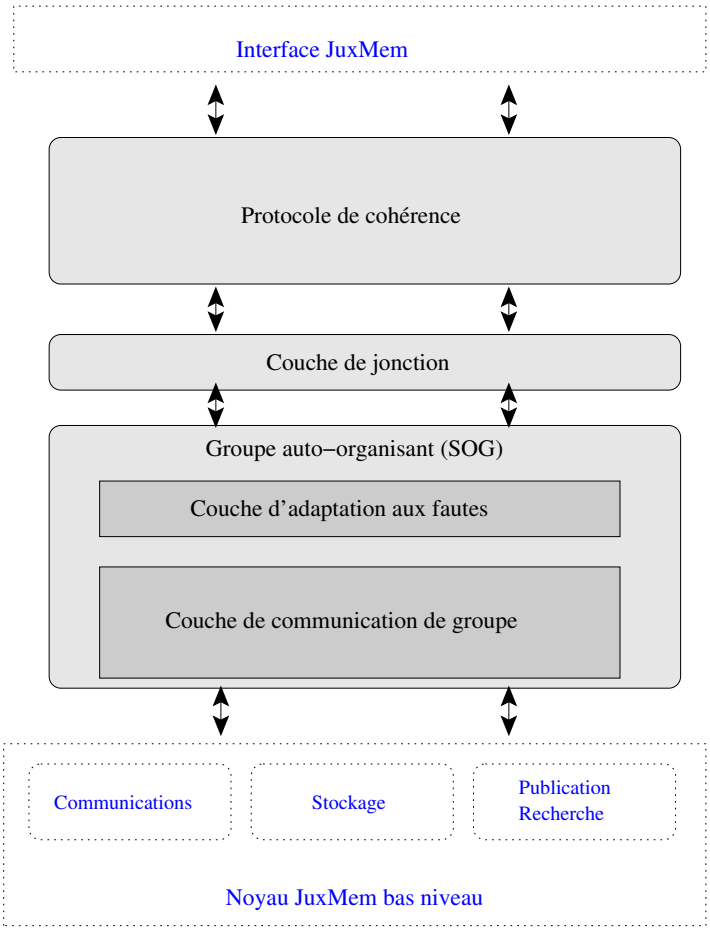

FIG. 6.3 – Une architecture en couches pour la gestion conjointe de la tolérance aux fautes et la cohérence des données.

La réplication employée pour améliorer la disponibilité. De nombreux mécanismes de tolérance aux fautes utilisent la réplication [98] pour supporter les défaillances des nœuds. Ainsi malgré la défaillance d'un des nœuds hébergeant une copie d'une donnée, cette donnée reste disponible grâce à l'existence des autres copies. Plusieurs techniques de réplication ont été étudiées et sont décrites en détail en section 3.3. Ces techniques présentent de différents compromis entre l'efficacité et les garanties.

La réplication employée pour améliorer la localité. Dans le cadre de la mise en œuvre de protocoles de cohérence, la réplication des données est utilisée pour améliorer les *performances* des accès aux données. De nombreux protocoles de cohérence utilisent la réplication afin de pouvoir placer des copies des données proches des processus qui y accèdent (voir chapitre 4). Elle permet à plusieurs processus de lire une même donnée concurremment en n'utilisant que des accès locaux. Cependant, lors de la mise à jour d'une donnée, le protocole doit actualiser ou invalider les autres copies de la donnée présentes dans le système de manière à maintenir la cohérence des différentes copies.

Dans les systèmes distribués, la réplication peut être employée à la fois à des fins de *localité* et de *disponibilité*. La question est de faire ou non la distinction entre les deux types de copies : celles créées pour améliorer la disponibilité et celles créées pour améliorer la localité. Il existe donc deux grandes familles de solutions : 1) une gestion découplée, et 2) une gestion intégrée.

Une gestion découplée. Il s'agit ici de faire une distinction stricte entre les copies créées pour la tolérance aux fautes et celles créées par le protocole de cohérence. Cette séparation permet un développement et une maintenance plus aisés mais sous-utilise le potentiel : le protocole de cohérence n'a pas connaissance des copies créées pour la tolérance aux fautes, ce qui implique qu'il ne peut pas les exploiter et réciproquement. En revanche, avec cette solution, il est possible d'ajouter un protocole de cohérence sans avoir à se préoccuper des problèmes liés à la tolérance aux fautes. De même, de nouveaux mécanismes de réplication à des fins de tolérance aux fautes peuvent être mis en place sans avoir à tenir compte du protocole de cohérence.

Une gestion intégrée. Cette approche consiste à gérer au même niveau les problèmes de la disponibilité et de la cohérence des données, aucune distinction n'est faite entre les différentes copies. Un seul ensemble de copies pour une donnée sert alors à la fois à améliorer sa localité et sa disponibilité. Par exemple, une copie créée sous le contrôle du protocole de cohérence à des fins de localité peut servir de sauvegarde en cas de fautes. De même, des copies créées par les mécanismes de tolérance aux fautes peuvent être utilisées par le protocole de cohérence. Notamment employée pour la conception de systèmes à MVP tolérants aux fautes, cette approche a pour principal avantage de permettre la mise en œuvre de protocoles efficaces [68] au prix d'un fonctionnement plus complexe. Le principal inconvénient de cette solution est que la maintenance et l'évolution du code sont délicates. La mise en place d'un nouveau protocole de cohérence, par exemple, nécessite de prendre en compte finement les mécanismes de tolérance aux fautes manipulant eux aussi les copies.

Nous souhaitons privilégier une gestion découplée afin de pouvoir : 1) implanter facilement de nouveaux protocoles de cohérence et aussi pouvoir répondre aux exigences d'un grand nombre d'applications ; 2) implanter de nouveaux mécanismes de réplication et aussi

pouvoir offrir divers compromis *garanties offertes/performances* ; et 3) avoir la possibilité de comparer des protocoles de cohérence en utilisant les mêmes mécanismes de tolérance aux fautes, ou de comparer différents mécanismes de tolérance aux fautes en mesurant les performances d'un même protocole de cohérence. Cependant, nous souhaitons également profiter des copies présentes au sein du système, que cela soit au niveau des protocoles de cohérence ou au niveau des mécanismes de tolérance aux fautes.

Malgré notre architecture découplée, nous verrons que *certaines* copies créées par le protocole de cohérence peuvent être exploitées par les mécanismes de tolérance aux fautes. De même *certaines* copies créées par les mécanismes de réplication à des fins de tolérance aux fautes peuvent être utilisées par le protocole de cohérence si elles se trouvent "bien" placées par rapport à un processus applicatif utilisateur.

Par exemple, au sein d'un groupe *cluster*, les mécanismes de tolérance aux fautes peuvent décider de créer une copie dans un groupe *cluster* distant afin de garantir la persistance de la donnée en cas de défaillance du groupe *cluster* entier. Cette défaillance peut être provoquée par exemple par une panne de courant de la grappe, les groupes *cluster* logiques représentant des grappes physiques. Un pair *client* situé justement dans le groupe *cluster* distant dans lequel a été répliquée la donnée pourra utiliser, via le protocole de cohérence, cette copie qui lui est proche.

Pour offrir ces propriétés, nous proposons l'architecture en couches représentée par la figure 6.3, et décrite ci-dessous.

6.2.2 La couche de communication de groupe

Cette couche est directement en relation avec le noyau JUXMEM. Son rôle est d'offrir un mécanisme de réplication *fiable*. Elle s'appuie à la fois sur les primitives de communication et celles de publication/recherche du noyau JUXMEM. Les primitives de communication sont utilisées pour échanger les messages entre les différents acteurs, pairs clients et pairs fournisseurs ; les primitives de publication/recherche, quant à elles, sont utilisées à des fins de localisation, par exemple pour retrouver les pairs fournisseurs.

Cette couche est composée de deux modules logiciels : un qui s'exécute sur les pairs de type fournisseur, et un qui s'exécute sur les pairs de type client. Côté fournisseur, cette couche est responsable de la gestion de la composition de groupe. Elle peut, par exemple, être instanciée par la pile logicielle présentée dans le chapitre précédent en section 5.4. Un groupe se voit attribuer un identifiant unique semblable à ceux donnés aux pairs, et c'est cet identifiant qui permet au client de communiquer avec le groupe. Côté client, cette couche offre une interface qui permet de communiquer avec un groupe de pairs de la même manière que l'on communiquerait avec un seul pair. La composition du groupe, c'est-à-dire le nombre et l'identité de chaque pair membre, est cachée aux couches supérieures côté client. L'interface offerte par cette couche permet donc aux pairs *clients* de communiquer avec un groupe de pairs, en l'occurrence un groupe de *fournisseurs*, de manière *transparente*, comme s'il n'y avait qu'un seul pair.

6.2.3 La couche d'adaptation aux fautes

La couche d'adaptation aux fautes offre la possibilité d'avoir des groupes *auto-organisants*. Cette couche est située juste au-dessus de la couche de communication de groupe et elle utilise des détecteurs de fautes du type décrit à la section 3.2.4 (réductibles à des détecteurs de fautes de classe $\diamondsuit P$). Son rôle est de gérer l'adaptation aux fautes du groupe. Lorsque l'un des membres du groupe est suspecté d'être défaillant par les détecteurs de fautes, cette couche va réagir en fonction 1) du rôle joué par le membre fautif (tous les membres d'un groupe ne jouent pas nécessairement le même rôle, voir chapitre 8) ; et 2) de la politique d'adaptation. Par exemple, il existe des mécanismes de communication de groupe basés sur la notion de *leader* ; si le *leader* est suspecté d'être fautif, une élection sera déclenchée afin de le remplacer. Les politiques d'adaptation peuvent être variées. Elles peuvent simplement consister à remplacer la copie suspecte afin de conserver un degré de réplication constant, mais elles peuvent également être plus complexes et par exemple tenir compte d'une estimation du *Mean Time Between Failures* (MTBF) courant afin de décider ou non de remplacer la copie suspecte. La couche d'adaptation aux fautes n'est présente que sur les pairs de type *fournisseur*.

Les couches vues jusqu'ici permettent la gestion de groupes de pairs qui vont recevoir les mêmes messages. Ceci nous offre des groupes de copies qui vont pouvoir être considérés comme stables sous certaines conditions de fautes. Les conditions varient selon l'implémentation de ces couches, et nous verrons au chapitre 8 qu'il en existe plusieurs. Ces groupes sont appelés par la suite des *groupes auto-organisants* ou SOG (pour l'anglais *Self-Organizing Group*). En effet, grâce à la couche d'adaptation aux fautes, les *SOG* vont continuer d'exister en présence de fautes, en se réorganisant.

6.2.4 Les protocoles de cohérence

Nous avons montré qu'il existe pour les systèmes à mémoire virtuellement partagée de nombreux modèles de cohérence implémentés par des protocoles de cohérence (voir section 4.1). À l'image du protocole de cohérence pris comme exemple dans notre étude de cas au chapitre 5, ces protocoles reposent souvent sur des entités implicitement supposées stables (comme un gestionnaire de page par exemple). Nous proposons de généraliser la solution vue à la section 5.4 et d'utiliser ces protocoles dans notre service de partage de données pour la grille en implémentant les entités supposées stables grâce à des groupes de copies auto-organisants (SOG). En effet, ces groupes peuvent être considérés comme stables si certaines conditions sont remplies. Nous avons vu que ces conditions varient selon l'implémentation choisie pour la couche de communication de groupe et selon la cardinalité du groupe. Par exemple pour notre étude de cas de la section 5.4, un nœud responsable d'une copie de référence est implémenté par un groupe auto-organisant. Ainsi l'automate représentant l'état du protocole de cohérence (objet de synchronisation, état des listes d'attentes, etc.) se trouve répliqué sur un ensemble de pairs. La pile logicielle proposée dans la section 5.4 pour mettre en œuvre la réplication permet de supporter $\lfloor \frac{n-1}{2} \rfloor$ fautes simultanées au sein d'un groupe, où n est la cardinalité du groupe.

Sur les pairs de type fournisseur, cette couche se situe au-dessus de la couche de communication de groupe. Elle se trouve donc répliquée. Cependant, cette réplication est invisible à ce niveau : pour chaque copie tout se passe comme si elle était le seul et unique exemplaire.

La couche de communication de groupe permet aux différentes copies de traiter les mêmes messages, dans le même ordre ; les copies évoluent donc de la même façon.

Sur les pairs de type client, le protocole de cohérence est en relation avec l'application par l'intermédiaire de la couche du noyau JUXMEM responsable des interactions avec l'application. Elle interagit avec les pairs fournisseurs (c'est-à-dire le groupe *data*) en utilisant la couche de communication de groupe.

6.3 Interactions entre le protocole de cohérence et les mécanismes de tolérance aux fautes

L'architecture décrite ci-dessus permet une gestion conjointe de la tolérance aux fautes et du protocole de cohérence. Cette architecture est découplée, permettant ainsi de séparer les problèmes liés à la tolérance aux fautes de ceux liés à la cohérence des données. Cependant, il apparaît que les connaissances du système ne sont pas les mêmes au sein des différentes couches. Au niveau du protocole de cohérence, la connaissance de l'architecture physique sous-jacente est très limitée. En effet, c'est au niveau des mécanismes de composition de groupe que l'on connaît individuellement les différents nœuds hébergeant une copie. En revanche, c'est le protocole de cohérence qui possède la meilleure connaissance de l'application car c'est à ce niveau que l'on prend en compte les différents pairs qui accèdent ou vont accéder à la donnée. Nous souhaitons donc que ces deux couches interagissent afin d'utiliser au mieux le potentiel offert par les ressources physiques utilisées par exemple lors de la création d'une nouvelle copie.

Création d'une nouvelle copie. En cas de faute, les mécanismes de réplication peuvent avoir à remplacer une copie suspectée. De même, l'intervention d'un nouvel utilisateur (pair accédant à la donnée) peut nécessiter la mise en place d'une nouvelle copie, voire d'un nouveau groupe de copies par le protocole de cohérence. Selon le cas, la création d'une nouvelle copie peut donc être décidée soit par les mécanismes de tolérance aux fautes, soit sous le contrôle du protocole de cohérence afin d'améliorer la localité pour un nouvel utilisateur.

Dans tous les cas, les mécanismes de réplication doivent en être informés. En effet, c'est à leur niveau que sont gérées les différentes copies. En revanche, le protocole de cohérence, plus proche de l'application, possédant notamment des listes d'attentes sur des verrous, est plus à même de choisir le pair devant héberger la nouvelle copie.

De plus, la création d'une nouvelle copie nécessite un transfert d'état du protocole de cohérence. En effet, le nouveau pair doit recevoir une copie de la donnée mais également l'état complet du protocole de cohérence : par exemple, un numéro de version attribué à la donnée, l'état de la donnée (verrouillée ou non), une ou plusieurs listes d'attente sur la donnée, etc.

Nous avons donc introduit une couche intermédiaire, appelée *couche de jonction*, responsable des interactions entre la couche contenant le protocole de cohérence et celles responsables de la tolérance aux fautes (SOG). C'est à ce niveau que l'on peut *ajuster* le degré d'interaction entre les deux. Le rôle de cette couche est décrit plus en détail au chapitre 8.

6.4 Vers une approche générique

Notre service de partage de données pour la grille, JUXMEM, doit être utilisable par un grand nombre d'applications, sur des plates-formes physiques potentiellement variées.

Cela implique que les différentes données gérées par JUXMEM doivent pouvoir être gérées selon différents compromis. En effet, les schémas d'accès aux données peuvent varier d'une donnée à l'autre, et certaines données sont plus *critiques* que d'autres.

Schémas d'accès aux données. JUXMEM est appelé à stocker différents types de données dont certaines ne seront accédées qu'en lecture. Par exemple, une matrice générée par un instrument de mesure sera en général stockée puis utilisée comme donnée d'entrée de nombreux calculs, mais non modifiée.

En revanche, d'autres données seront accédées presque uniquement en écriture, comme des points de reprises sauvegardés par exemple. Ces derniers sont remplacés au fur et à mesure qu'ils deviennent obsolètes et ne sont accédés en lecture qu'en cas de retour en arrière de l'application qui les a générés, ce qui est généralement peu fréquent.

La majorité des données n'ont pas des schémas d'accès aussi simples. Elles sont accédées aussi bien en lecture qu'en écriture. Les accès peuvent également être concurrents. Il est alors important de mettre à la disposition des applications des protocoles de cohérence leur permettant de faire de manière efficaces les accès aux données.

Selon le schémas d'accès aux données d'une application, certains modèles et protocoles de cohérence seront plus efficaces que d'autres. Par exemple, dans le cas du modèle de cohérence à l'entrée décrit dans la section 5.2, l'association d'un objet de synchronisation à chaque donnée partagée afin de mettre en place des accès exclusifs à la donnée n'est d'aucune utilité pour le partage d'une donnée en lecture seule.

De plus, les applications sont toutes implémentées en supposant un certain modèle de cohérence. JUXMEM doit donc offrir une palette de protocoles de cohérence implémentant les divers modèles supposés par les applications qui vont l'utiliser. Cela permet de limiter l'impact de l'utilisation de notre service de partage de données au niveau du code des applications car l'implémentation des accès aux données peut varier selon le modèle de cohérence. Mais cela permet également de conserver le modèle de cohérence initialement choisi par les concepteurs des applications.

Criticité des données. Les données se différencient également par leur niveau de criticité. En effet, certaines données peuvent être considérées comme *très critiques* alors que d'autres peuvent éventuellement être perdues, par exemple parce qu'il est facile de les régénérer.

Des données générées par un instrument de mesure, par exemple, ne doivent pas être perdues. En effet, leur régénération peut s'avérer très coûteuse, voire impossible.

En revanche certaines données sont des données temporaires qui sont générées à partir d'algorithmes simples, comme le résultat du découpage d'une matrice. Si une sous-matrice est perdue, elle pourra facilement être régénérée par l'algorithme de découpage.

De ce fait, il nous semble important de prendre en compte le niveau de criticité des données : garantir la persistance d'une donnée a un coût en termes de performance d'accès. Plus les hypothèses sur les fautes sont pessimistes, plus le coût des mécanismes de réplication

répercuté sur les temps d'accès à la donnée va être important. L'application doit pouvoir choisir le compromis *niveau de risque/performance* qu'elle accepte pour chaque donnée gérée par le service.

Des blocs interchangeables. JUXMEM doit s'adapter aux exigences des applications en termes de performance et de garantie ainsi qu'aux contraintes physiques, notamment en terme de probabilité d'occurrence de fautes, pour chacune des données gérées par le service. Chacune des couches décrites ci-dessus doit donc présenter plusieurs implémentations.

Cela signifie que les interfaces entre les couches doivent supporter un grand nombre de combinaisons protocoles de cohérence/mécanismes de tolérance aux fautes. De plus, le choix de la combinaison optimale doit pouvoir être fait pour chaque donnée. Des mécanismes d'instantiation dynamiques ont donc été mis en place.

Ces interfaces, ainsi que cette architecture en couches, permettent de pouvoir implémenter facilement de nouveaux protocoles de cohérence, de nouveaux mécanismes de réplication, et d'expérimenter les différentes combinaisons. De plus cela permet aux applications de pouvoir elles-mêmes, pour chaque donnée partagée, choisir le compromis qui convient le mieux en fonction du niveau de risque estimé ainsi que des performances attendues. Des valeurs par défaut sont bien entendu fournies de manière à permettre une utilisation aisée.

La conception du cœur des couches doit prendre en compte les contraintes imposées par l'architecture physique des grilles de calcul afin d'offrir aux applications des temps d'accès aux données partagées performants. C'est l'objet du chapitre suivant.

Chapitre 7

Gestion hiérarchique de la réplication et de la cohérence des données

Sommaire

Le chapitre précédent décrit notre proposition d'architecture en couches pour une gestion conjointe de la tolérance aux fautes et de la cohérence des données. Le principal but de cette architecture est de faciliter la mise en œuvre de nouveaux protocoles de cohérence ainsi que de nouveaux mécanismes de tolérance aux fautes.

Lors de la mise en œuvre de protocoles de cohérence et de mécanismes de tolérance aux fautes il est important de prendre en compte les caractéristiques physiques des grilles de calcul pour offrir des accès performants aux données partagées.

Ce chapitre décrit comment il est possible concevoir les couches de notre architecture de manière à construire des protocoles de cohérence et des mécanismes de tolérance aux fautes prenant en compte les caractéristiques des grilles. Nous généralisons ici les solutions abordées au chapitre 5 pour faire face aux *problèmes liés au passage à l'échelle* : nous proposons

une approche *hiérarchique* pour la conception des différentes couches décrites au chapitre précédent.

7.1 Gestion hiérarchique de la cohérence

Au chapitre 5, nous avons montré sur un exemple de protocole de cohérence qu'une conception hiérarchique du protocole permet d'adapter ce protocole à la topologie réseau de la grille telle que décrite à la section 2.1. Nous généralisons ici cette solution.

7.1.1 Limites d'un protocole non-hiérarchique

Lorsque l'on étudie les protocoles de cohérence existants, par exemple dans le cadre des mémoires virtuellement partagées, on s'aperçoit qu'ils ne conviennent pas à la topologie des grilles de calcul. En effet, la plupart d'entre eux se base sur une topologie supposée plate. De plus, comme détaillé dans la section 4.1, de nombreux protocoles exploitent les notions de *gestionnaire de donnée*, de *copie de référence d'une donnée*, etc. Il en résulte que chaque accès à une donnée nécessite des communications avec un nœud particulier (avec le nœud hébergeant la copie de référence dans l'exemple du chapitre 5).

Au sein d'une architecture de type grille, tous les liens réseau ne présentent pas les mêmes caractéristiques. Si l'on reprend la description des grilles de calcul vues comme des fédérations de grappes du chapitre 2.1, on peut observer une différence notable (de l'ordre d'un facteur 1.000, voire 10.000) entre la latence des liens au sein d'une grappe et celle des liens reliant les grappes entre elles. Par exemple, au sein de grappes équipées de réseaux de type *Myrinet* [119], la latence entre deux nœuds sera de l'ordre de 2 μs, alors que la latence entre deux nœuds situés dans chacune de ces grappes sera de l'ordre de quelques dizaines de *millièmes de secondes* si l'on utilise un réseau comme *Renater* [125] voire plusieurs centaines de *millièmes de secondes* si elles sont reliées par Internet. De plus, dans le cas de grappes reliées par un réseau tel Internet, cette différence concerne également le débit des liens réseau et non seulement la latence.

Nous avons vu au chapitre 5 que si des nœuds géographiquement éloignés partagent une même donnée, de nombreuses communications vont utiliser des liens "longue distance" (intergrappe). Un protocole de cohérence non-hiérarchique risque d'être inefficace

Étant donné la topologie réseau, il nous semble judicieux d'adapter le protocole de cohérence afin de tenir compte de la structure hiérarchique sous-jacente. En effet il est souhaitable de limiter l'utilisation des liens intergrappe à plus forte latence.

7.1.2 Une vision hiérarchique

Il est difficile d'envisager une application scientifique à très grande échelle impliquant plusieurs milliers de processus pour laquelle chaque processus communique indifféremment avec n'importe quel autre processus. Dans le cas des applications de couplage de codes auxquelles nous nous intéressons (voir section 2.2), l'ensemble des processus impliqués est partitionnable naturellement en sous-groupes. Les processus membres d'un même sous-groupe exécutent généralement un code parallèle et les différents codes s'exécutant sur les différents sous-groupes sont couplés, c'est-à-dire qu'ils échangent de temps en temps des

informations. Il en résulte qu'il existe de nombreuses communications aux sein de chaque sous-groupe dans lesquels les processus sont fortement liés et seulement quelques-unes entre les sous-groupes coopérants.

Déploiement des applications. Lors du déploiement de telles applications, à des fins d'efficacité, il est judicieux de placer les processus appartement à un même groupe sur des nœuds proches les uns des autres. Ici, la distance se mesure en *latence* réseau entre les nœuds. Typiquement chaque groupe de processus d'une application de type *couplage de codes* va être placé sur une même grappe. Ainsi, les entités fortement liées (c'est-à-dire partageant de nombreuses données) vont bénéficier de communications efficaces alors que les liens reliant entre eux les différents groupes de nœuds, sur lesquels s'exécutent les groupes de l'application de couplage de code, présenteront généralement des latences plus élevées (voir section 2.2).

7.1.3 Des protocoles de cohérence hiérarchiques

Lors de notre étude de cas, au chapitre 5, nous avons montré les faiblesses d'un exemple de protocole de cohérence plat. En nous inspirant de *Clustered Lazy Release Consistency* (CLRC [6]) qui propose d'améliorer la localité en créant des caches locaux et de [5] qui propose une approche hiérarchique pour la gestion distribuée d'objets de synchronisation, nous avons proposé une solution. Elle consiste à hiérarchiser le protocole de cohérence afin d'améliorer ses performances en limitant l'utilisation des liens intergrappe. À cette fin nous avons introduit la notion de *copie de référence locale*.

Le protocole de cohérence étudié dans le chapitre 5 présente une caractéristique commune à de nombreux protocoles de cohérence : un nœud joue un *rôle central* et est responsable des accès à la donnée. Cette caractéristique se retrouve au niveau de tous les protocoles dits *home-based* ou ceux utilisant un gestionnaire de page. Notre solution décrite sur un exemple au chapitre 5 est généralisable à tous les protocoles présentant cette caractéristique.

Le principal intérêt de la construction hiérarchique du protocole de cohérence est de permettre une différenciation des accès selon les grappes et ainsi de pouvoir favoriser certains accès. Par exemple si une donnée est partagée par plusieurs processus répartis dans des grappes différentes, le protocole de cohérence va chercher à favoriser les accès locaux par rapport aux accès externes (provenant d'une autre grappe). Cela va permettre de limiter le cas où la donnée est accédée dans une grappe A, puis dans une grappe B pour revenir à la grappe A et ainsi de suite. Dans les cas où cela est possible, les accès se feront ainsi d'abord dans la grappe A puis dans la grappe B, limitant ainsi l'utilisation des liens intergrappe dont la latence est coûteuse pour propager les mises à jour de la donnée partagée.

Cependant, il est possible qu'au niveau applicatif, ces "allers-retours" de la dernière version de la donnée soient obligatoires. Dans ce cas ils se feront naturellement grâce aux synchronisations présentes au niveau applicatif. De plus, afin d'éviter une famine, il est nécessaire de borner le nombre de fois où les accès locaux sont privilégiés. Sinon, il serait possible qu'une grappe (ou un ensemble de nœuds au sein d'une grappe) soit bloquée et il en résulterait une séquentialisation de l'application dont le coût pourrait être supérieur au gain offert par la hiérarchisation du protocole de cohérence.

Il existe un deuxième avantage offert par la hiérarchisation du protocole de cohérence. Cela permet d'avoir une vision restreinte : même si une donnée est accédée par de très nom-

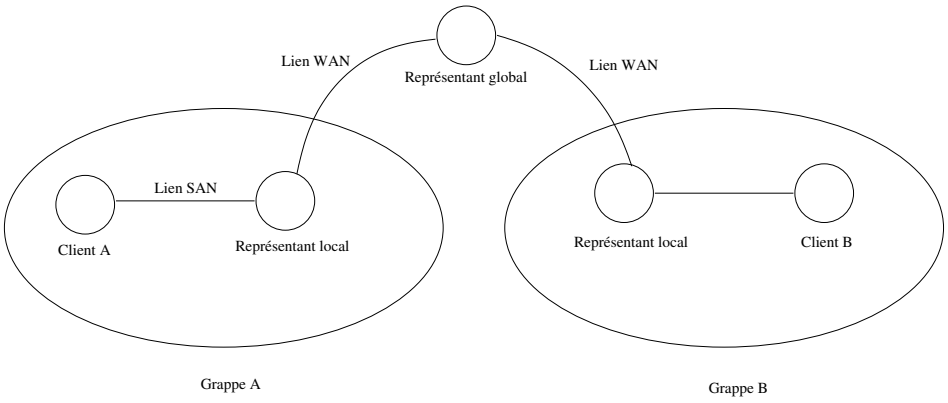

FIG. 7.1 – Un protocole de cohérence hiérarchique.

breux nœuds, une *copie de référence locale* n'interagira qu'avec un sous-ensemble limité, l'ensemble des nœuds de la grappe partageant la donnée associée à la copie de référence. De même, la copie de référence globale d'une donnée ne sera accédée que par les copies de références locales de cette donnée situées dans les grappe dans lesquelles se s'exécutent des clients partageant cette donnée. La hiérarchie permet donc de limiter le nombre de clients potentiels d'une copie de référence, locale ou globale. Il en résulte une diminution des accès sur les copies de référence ainsi qu'une réduction de la taille des listes d'attente. Cela permet un meilleur passage à l'échelle du protocole de cohérence.

De la même manière que nous avons introduit la notion de *copie de référence locale* à la section 5.3, il est possible de généraliser en introduisant la notion de *représentant local*. Nous avons remarqué dans la section 4.1 que de nombreux protocoles de cohérence sont fondés sur des entités fixes jouant un rôle central, les nœuds gestionnaires par exemple. Les représentants locaux peuvent jouer ce rôle vis-à-vis des clients situés dans le même groupe *cluster*. Au sein de chaque groupe *cluster* contenant des nœuds partageant une donnée d, le protocole de cohérence place un *représentant local* pour cette donnée d (une *copie de référence locale* par exemple). Chaque fois que le protocole de cohérence s'exécutant sur un nœud client doit accéder à l'entité jouant le rôle central du protocole, il contacte le représentant local pour cette donnée au sein de sa grappe. Ainsi, les protocoles de cohérence client n'ont pas de vision hiérarchique. De leur point de vue, tout se passe comme si le représentant local auquel ils accèdent était l'entité jouant le rôle central du protocole de cohérence.

Cependant les données peuvent être partagées par des nœuds n'appartenant pas à une même grappe. Les modifications des uns doivent pouvoir être vues par *tous* les autres. Aussi pour chaque donnée partagée, le protocole de cohérence doit également gérer la cohérence entre les différents représentants locaux répartis dans plusieurs grappes d'une fédération. Nous aboutissons à un protocole de cohérence *hiérarchique* à deux niveaux illustré par la figure 7.1. Au niveau *local*, les nœuds d'une même grappe accèdent à une donnée partagée via un *représentant local* situé dans cette même grappe. Au niveau *global*, les *représentants locaux* accèdent à un *représentant global* qui joue le rôle central du protocole de cohérence vis-à-vis des *représentants locaux*. Les *représentants locaux* se comportent comme des clients vis-à-vis du *représentant global*.

Ainsi lorsqu'un nœud a besoin d'accéder à une donnée partagée, il contacte le représen-

tant local qui peut soit permettre l'accès directement, soit avoir à contacter le représentant global afin de se mettre à jour ou de demander un accès exclusif à la donnée par exemple.

La hiérarchisation du protocole permet d'améliorer les performances des accès aux données partagées sur les architectures de type grille cependant cela ne permet pas de supporter les fautes. En effet, la faute d'un nœud jouant le rôle de représentant (local ou global) du protocole de cohérence peut entraîner la perte de la donnée[1]. La section suivante présente des mécanismes de tolérance aux fautes adaptés à la grille et aux protocoles de cohérence *hiérarchiques* décrits dans cette section.

7.2 Gestion hiérarchique de la tolérance aux fautes

De même que pour la gestion de la cohérence des données, les mécanismes de tolérance aux fautes doivent prendre en compte les différences de latence entre les liens intergrappe et intragrappe. Nous proposons donc une approche *hiérarchique* pour la gestion de la tolérance aux fautes. Nous utilisons les détecteurs de fautes basés sur une approche hiérarchique qui ont été conçus au cours de la thèse de Marin Bertier [16]. Ces détecteurs sont décrits dans la section 3.2.4. Nous avons également mis en place des mécanismes de réplication hiérarchiques.

7.2.1 Détecteurs de fautes basés sur une approche hiérarchique

Les mécanismes de tolérance aux fautes que nous avons conçus utilisent des détecteurs de fautes. Ils sont en effet nécessaires aussi bien pour la prévention des fautes (la réplication) que pour la réaction aux fautes (rôle de la couche d'adaptation aux fautes).

Accord. Les mécanismes de prévention de fautes comme la réplication peuvent utiliser des algorithmes d'accord afin d'offrir les garanties nécessaires en cas de fautes, comme permettre une mise à jour atomique de plusieurs copies pour qu'il reste des copies à jour en cas de fautes. Les détecteurs de fautes permettent de résoudre des problèmes d'accord en présence de fautes.

Attente infinie. De manière plus générale, les détecteurs de fautes permettent d'éviter d'attendre infiniment la réponse d'un nœud fautif. Par exemple, nous avons vu au chapitre 5 que les mécanismes de réplication peuvent utiliser des algorithmes de consensus. Ces derniers doivent être prévenus en cas de fautes afin de pouvoir prendre une décision sans attendre les nœuds fautifs.

Réaction/adaptation aux fautes. Il est nécessaire de prendre en compte les fautes qui surviennent et de réagir afin de pouvoir supporter de nouvelles fautes dans le futur. C'est le rôle joué par la couche d'adaptation aux fautes, permettant de mettre en place les *groupes auto-organisants*. À cette fin, les détecteurs vont permettre de signaler aux algorithmes d'adaptation aux fautes qu'un nœud est suspecté d'être défaillant.

[1] Nous considérons ici que la perte de la dernière version de la donnée, qui peut être détenue par un des représentants locaux, est équivalente à la perte de la donnée.

Les détecteurs de fautes utilisés permettent d'enregistrer une fonction qui sera appelée en cas de suspicion. Lors de l'enregistrement de cette fonction il est possible de stipuler le niveau de qualité de service que l'on souhaite en terme de compromis entre la réactivité des détecteurs et le taux de fausses détections (suspicion d'un nœud non fautif).

Parmi les caractéristiques de ces détecteurs de fautes, celles qui nous intéressent particulièrement sont :

- le passage à l'échelle,
- l'adaptabilité,
- le paramétrage du niveau de qualité de service.

Le passage à l'échelle. Comme détaillé au chapitre 3.2.4, ces détecteurs sont hiérarchiques et s'adaptent donc très bien à une architecture de type grille. D'autant mieux à une fédération de grappes. Ces détecteurs sont basés sur des mécanismes d'émissions régulières de messages de vie. En associant les groupes locaux des détecteurs aux groupes *cluster* JUXMEM comme illustré par la figure 7.2, on obtient des échanges de messages de vies de type "tous-vers-tous" aux sein de chaque groupe *cluster* et des échanges de messages de vie entre représentant de grappes.

L'adaptabilité. Les architectures de type grille sont dynamiques par nature. La charge des différents nœuds, des passerelles, des *switches* peut varier au cours du temps. Cela est notamment dû au fait que les grilles de calcul sont généralement multi-utilisateur. Les détecteurs de fautes utilisés prennent en compte ces variations dans le calcul des estimations des prochaines arrivées de messages de vie et offrent donc une bonne précision.

Le paramétrage du niveau de qualité de service. Les détecteurs de fautes vont être utilisés à de multiples niveaux de notre architecture en couche. Notamment 1) au niveau des mécanismes de réplication, pour de leur éviter d'attendre indéfiniment des messages de nœuds fautifs ; et 2) au niveau de la couche d'adaptation aux fautes, afin de réagir en cas de faute (en remplaçant les nœuds fautifs par exemple). Ces différents cas d'utilisation ne nécessitent pas les mêmes niveaux de qualité de service de la part des détecteurs. En effet, au niveau des couches basses, il va être important d'avoir une détection de faute réactive pour permettre à un algorithme de consensus de progresser. Ici, un cas de fausse détection n'est pas critique. En revanche au niveau des couches hautes, comme par exemple la couche d'adaptation aux fautes, la réactivité semble moins importante que le risque de fausse détection. Une détection de faute remontée à ce niveau va effectivement déclencher un lourd processus de réparation. Dans le cas où la politique d'adaptation est de conserver le degré de réplication, cela va automatiquement déclencher la recherche d'un nœud remplaçant ainsi que l'insertion de ce nœud dans le groupe de copies et l'initialisation de cette nouvelle copie (transfert d'état)[2]. Nous utilisons donc la possibilité de régler la qualité de service des détecteurs de fautes afin d'avoir à chaque niveau une détection adaptée.

Les détecteurs de fautes proposent un service de détection à deux niveaux : local et global. La détection au niveau global se fait par l'intermédiaire de *mandataires* sélectionnés au sein de chaque groupe local.

[2]Un exemple de mécanisme de réparation est donné au chapitre 9.

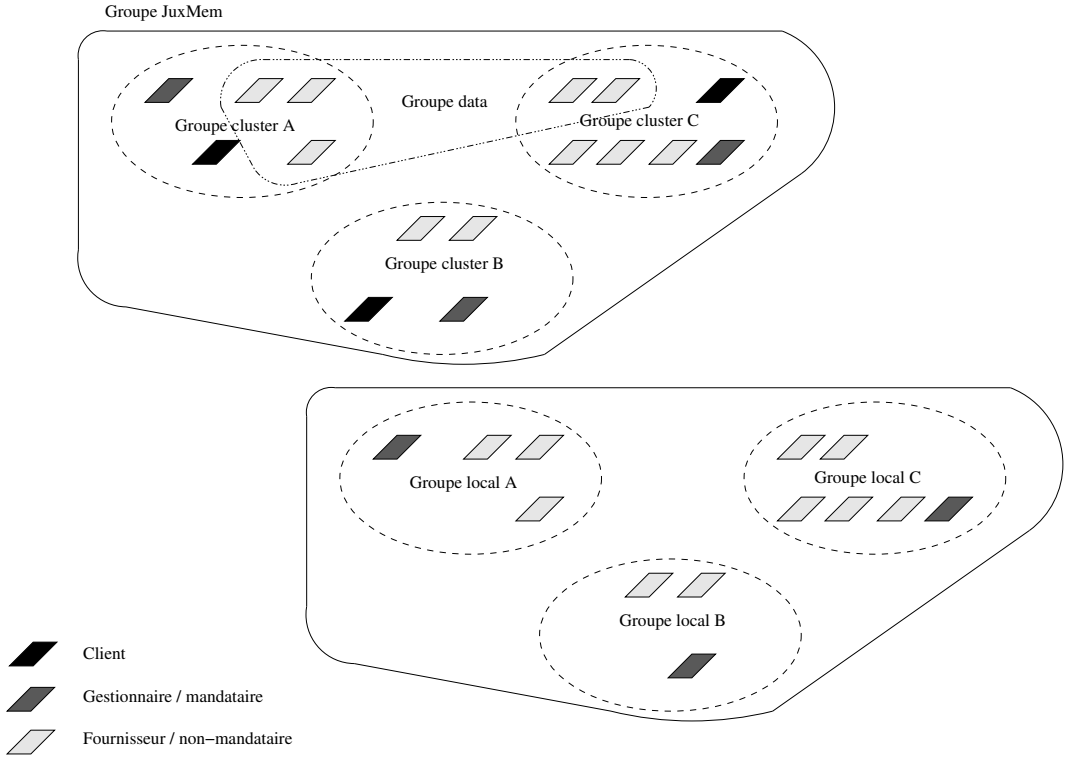

FIG. 7.2 – Correspondance entre les groupes de détection de fautes et les groupes *cluster* JUXMEM.

Notre architecture a été conçue de manière à pouvoir intégrer les détecteurs de fautes en faisant correspondre les groupes locaux des détecteurs avec les groupes *cluster* de JUXMEM. Dans chaque groupe local/groupe *cluster* le mandataire initial est le pair *gestionnaire* du groupe *cluster*. La figure 7.2 illustre cette correspondance. Nous allons voir dans la suite de cette section que cette correspondance permet de ne gérer, au niveau de chaque groupe *cluster*, que les défaillances des pairs situés au sein du même groupe *cluster* ; et au niveau du groupe JUXMEM, de ne gérer que les défaillances des groupes *cluster* (c'est-à-dire les défaillances de grappes).

7.2.2 Protocole hiérarchique de composition de groupe

Il existe de multiples travaux sur les mécanismes de composition de groupe, il sont décrits à la section 3.3. Cependant, la gestion de ces groupes se base généralement sur des mécanismes utilisant des synchronisations globales. La taille de ces groupes doit donc être restreinte. De plus, il est préférable que les délais de communication entre les membres d'un même groupe soient courts car si les temps de synchronisation se rapprochent des temps interfaute, les synchronisations vont souvent échouer.

Les mécanismes de composition de groupe pair-à-pair passent à l'échelle tout en relâchant les contraintes. Ils n'offrent généralement pas de garanties : ils sont dits "*best effort*". Or, pour la réplication des entités des protocoles de cohérence, il est nécessaire d'offrir des garanties. Sans cela, différentes copies du protocole de cohérence évolueraient de manière différente, aboutissant à une incohérence au niveau du partage de donnée, par exemple la lecture d'une valeur obsolète.

Le groupe que nous souhaitons gérer est le le groupe *data*, qui s'étend potentiellement sur plusieurs grappes. Nous souhaitons également pouvoir offrir des garanties strictes d'un point de vue tolérance aux fautes et cohérence de données. Nous proposons donc une solution permettant d'allier *passage à l'échelle* et *garanties* pour la réplication des entités du protocole de cohérence.

Une hiérarchie imposée. Le but de la réplication est de ne pas perdre la donnée ni l'état du protocole de cohérence, y compris en cas de fautes. Le chapitre 5 décrit comment il est possible d'atteindre ce but sur un exemple de protocole de cohérence à l'entrée en mettant en place des mécanismes pour répliquer la copie de référence. Cette solution est généralisable à tous les protocoles présentant la caractéristique d'avoir un nœud jouant un *rôle central* (voir section 7.1.3).

Dans le cas d'un protocole de cohérence hiérarchique, comme décrit à la section 7.1.3, il est nécessaire également de répliquer des *représentants locaux*. En effet, un représentant local peut être à un moment donné le seul à héberger la dernière version de la donnée (dans le cas par exemple où un accès en écriture vient de se produire dans la grappe dans laquelle il se situe).

Les entités que nous souhaitons répliquer sont les entités *critiques* du protocole de cohérence : chaque représentant local et le représentant global de chaque donnée.

Au sein d'une grappe, les algorithmes de réplication peuvent bénéficier d'un réseau performant présentant une faible latence et un haut débit. Les mécanismes de réplication ba-

sés sur des synchronisations globales, comme ceux utilisés dans l'exemple du chapitre 5, semblent donc appropriés pour la réplication des représentants locaux.

Le représentant global (la copie de référence globale dans l'exemple du chapitre 5) est une entité particulièrement critique. C'est notamment à ce niveau que les accès aux grappes sont distribués. En cas de perte de cette entité, les représentants locaux associés se retrouveraient bloqués ainsi que leurs clients. De plus, pour chaque donnée il est nécessaire de tolérer la faute d'une grappe complète car ce genre de faute est possible au sein d'une grille de calcul, comme expliqué à la section 2.4. Il est donc nécessaire que le représentant global du protocole de cohérence soit répliqué sur plusieurs grappes. Afin de limiter le nombre de nœuds impliqués dans la gestion de chaque donnée, la copie de référence globale est représentée par l'ensemble des pairs fournisseurs fournissant une copie de la donnée au sein du groupe JUXMEM.

Définition 7.1 : ***local data group*** — (LDG). Un LDG est un groupe de pairs de type fournisseur situés dans un même groupe *cluster* et possédant une copie d'une même donnée. Il est le *représentant local* du protocole de cohérence gérant la donnée au sein du groupe *cluster* dans lequel il se trouve (par exemple, la copie de référence locale pour un protocole de cohérence *home-based*). Il correspond à l'intersection du groupe *data* d'une donnée et d'un groupe *cluster*.

Définition 7.2 : ***global data group*** — (GDG). Un GDG est un groupe dont les membres sont les LDG représentant une même donnée.

Chaque donnée partagée allouée au sein de JUXMEM se voit donc associer un GDG contenant un ou plusieurs LDG. Le protocole de cohérence hiérarchique est implanté au-dessus de la hiérarchie LDG/GDG comme illustré par la figure 7.3. Si une donnée n'est accédée que dans un seul groupe *cluster*, l'unique LDG associé est confondu avec le GDG. Il est également possible dans ce cas de placer des copies dans un autre groupe *cluster*, créant ainsi un deuxième LDG afin de pouvoir tolérer la panne de la grappe.

7.2.3 Propagation des messages

Nous venons de voir que nous utilisons les groupes pour répliquer des entités des protocoles de cohérence. Les protocoles de cohérence s'exécutant sur les clients doivent pouvoir communiquer avec ces entités. En effet, cela est nécessaire lors des accès à la donnée, comme c'est le cas pour le protocole de cohérence décrit au chapitre 5.

Les pairs de type client communiquent avec le LDG présent dans leur grappe. De même, les LDG (représentants locaux des protocoles de cohérence) communiquent avec le GDG (le représentant global du protocole de cohérence). Par exemple lorsqu'un client souhaite obtenir un accès à la donnée alors que cet accès est déjà autorisé dans un groupe *cluster* différent, cela va engendrer des communications : 1) entre le client et son LDG[3] ; et 2) entre les LDG et le GDG afin que l'autorisation d'accéder à la donnée puisse changer de groupe *cluster*. Ces communications sont détaillées ci-dessous.

[3] *Son* LDG représente le LDG situé dans le même groupe *cluster*.

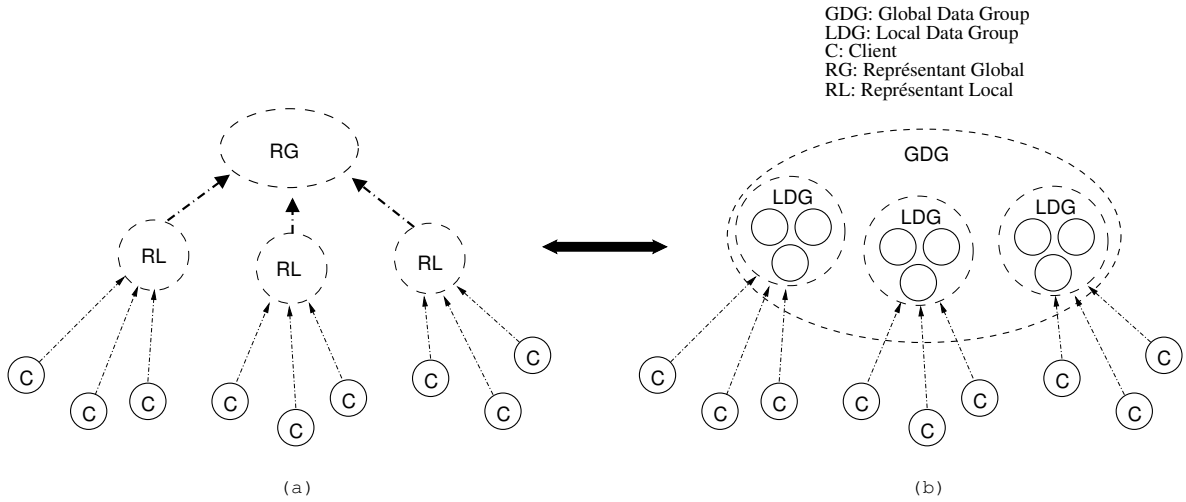

FIG. 7.3 – Réplication des entités critiques du protocole de cohérence hiérarchique.

Communication avec un LDG. Le protocole de cohérence client communique avec le LDG à travers la couche de communication de groupe client. Cette dernière est en charge de localiser[4] le LDG. Selon l'implémentation, il peut s'agir de localiser un représentant *non fautif* du groupe, plusieurs membres ou tous les membre du LDG. Quand le LDG est localisé, le message est envoyé. S'il s'agit d'une communication de groupe, la couche de jonction ajoute l'étiquette GRP_COMM et le message est transmis à tous les membres du LDG en utilisant un mécanisme de diffusion atomique. Dans le cas contraire, un seul membre est contacté. C'est le cas pour des opérations n'impliquant pas de mise à jour de l'état du protocole de cohérence des données.

Communication avec un GDG. Le protocole de cohérence serveur de niveau local (LDG) communique avec le protocole de cohérence de niveau global (GDG) de la même manière que les clients le font avec les LDG. Ceci implique que sur chaque pair de type fournisseur et pour chaque donnée stockée, sont instanciés :

1. les couches de communication de groupe ;
2. le protocole de cohérence local représentant le LDG ;
3. le protocole de cohérence global représentant le GDG.

Lorsqu'un message est envoyé au GDG, par exemple une demande d'accès exclusif (verrouillage), le message est retransmis à l'ensemble de ses membres (c'est-à-dire les LDG). Les LDG recevant le message le retransmettent à l'ensemble de leur membres (les pairs fournisseurs). Cependant, il ne s'agit là pas d'un simple mécanisme de diffusion hiérarchique. En effet de manière à maintenir la cohérence des copies, les messages impliquant une mise à jour (ici une mise à jour de la liste d'attente pour un accès exclusif) doivent être ordonnés. Il ne faut pas que des mises à jour concurrentes soient appliquées dans des ordres différents. Cela permet d'assurer que l'accès exclusif ne sera accordée qu'à un seul client à la fois. Le

[4]Ici localiser signifie trouver un identifiant. La localisation au sens réseau (adresse IP, port) est effectuée par les mécanismes JXTA via le noyau JUXMEM.

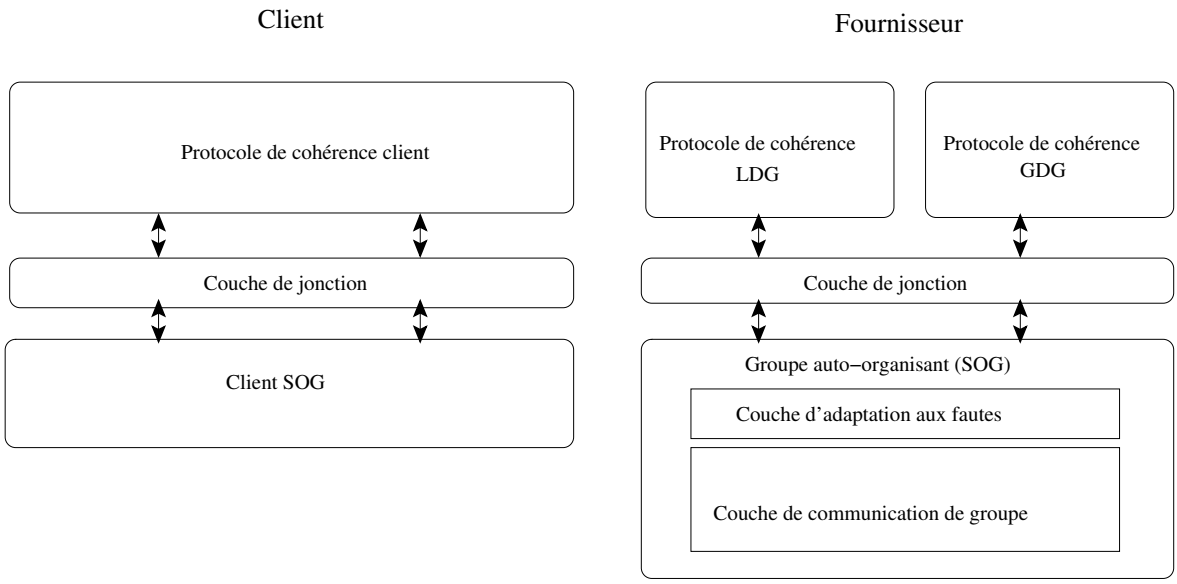

FIG. 7.4 – Architecture en couches hiérarchique.

chapitre 8 détaille des mécanismes permettant de garantir une diffusion *atomique* au sein des GDG et LDG.

7.3 Un exemple de scénario

Nous illustrons ici le fonctionnement des différentes couches présentées au chapitre 6 implémentées de manière hiérarchique (figure 7.4). Un exemple détaillé de mise en œuvre se trouve au chapitre 8.

Fonctionnement sans fautes.

Un client alloue une donnée, le noyau JUXMEM instancie le protocole de cohérence client en lui transmettant la liste des fournisseurs qu'il a sélectionnés. Avant de s'instancier, le protocole client envoie des messages contenant des requêtes d'allocation aux pairs fournisseurs contenant une liste de listes de fournisseurs (une liste par groupe *cluster*). À la réception de cette requête les pairs fournisseurs instancient les protocoles de cohérence de niveau LDG et GDG, la couche de jonction et la couche de groupe auto-organisant pour cette donnée. Lorsque les différentes couches sont instanciées et prêtes à fonctionner, elles publient des annonces à la fois dans leur groupe *cluster* et dans le groupe JUXMEM. Ces annonces permettront aux clients de faire l'association groupe *data* /donnée. Elles contiennent en effet l'identifiant de la donnée ainsi que l'identifiant (ou les identifiants selon le type de réplication utilisé) à utiliser pour communiquer avec le groupe. Pour finir, un acquittement est retourné au client. Lorsque le client a reçu les acquittements de la part des fournisseurs, les couches de jonction et de communication de groupe clientes sont instanciées, puis l'appel à la méthode

d'allocation se termine en retournant l'identifiant de la donnée[5].

Par la suite, le protocole de cohérence client interagit avec le protocole de cohérence de niveau LDG via la couche de communication de groupe. De même, le protocole de cohérence de niveau LDG communique avec le protocole de cohérence de niveau GDG via la couche de communication de groupe.

Afin de partager la donnée avec d'autres clients, l'identifiant de la donnée doit être connu. L'architecture en couches client peut donc également être instanciée sans qu'il y ait interaction avec des pairs fournisseurs. Dans ce cas, la couche client de communication de groupe est responsable de localiser le LDG avec lequel elle doit communiquer. Pour cela elle recherche les annonces publiées par les groupes auto-organisants (SOG pour *Self-Organizing Group*) dans le groupe *cluster* ce qui lui permet de trouver le LDG présent dans le même groupe *cluster*. Si la recherche échoue, une nouvelle recherche est effectuée au niveau du groupe JUXMEM afin de trouver le GDG.

Rôle de l'interface entre protocole de cohérence et couche de tolérance aux fautes. Lorsqu'il est nécessaire de créer une nouvelle copie, pour conserver le degré de réplication après l'occurrence d'une faute par exemple, une interaction entre les couches de communication de groupe et celles du protocole de cohérence est nécessaire afin d'initialiser cette nouvelle copie. Mais le rôle de cette interface n'est pas limité à cela. C'est à ce niveau que l'on peut notamment optimiser les performances : on souhaite d'une part que la réplication soit invisible au niveau des protocoles de cohérence et d'autre part que les mécanismes de communication de groupe soient indépendants du protocole de cohérence. La couche de jonction au niveau du client est responsable de déterminer si un message doit être envoyé au groupe ou à un seul de ses membres. En effet, certains messages des protocoles de cohérence ne nécessitent pas de modification de l'état du protocole de cohérence côté fournisseur et peuvent donc être traités par n'importe quel membre du groupe. Ce cas est détaillé dans l'exemple décrit au chapitre 8. Ainsi, la couche de jonction permet d'éviter des communication de groupe coûteuses et inutiles.

Gestion des fautes. Notre vision hiérarchique permet de distinguer deux types de faute : les pannes franches de nœuds au sein d'un groupe *cluster*, et les pannes franches de grappes (groupes *cluster*). Cela permet de ne pas se préoccuper au sein d'un groupe *cluster* des défaillances de nœuds pouvant intervenir au sein des autres groupes *cluster* du groupe JUXMEM. En cas de suspicion de la part des détecteurs de fautes côté fournisseur, la suspicion est remontée au niveau de la couche d'adaptation aux fautes. Elle est filtrée (seules les défaillances des membres du groupe sont pertinentes à ce niveau) puis prise en compte. La prise en compte peut consister en la création d'une nouvelle copie en cas de faute d'un pair fournisseur ou en l'allocation d'un nouveau LDG en cas de faute d'une grappe entière. De nouvelles publications d'annonces sont également faites afin de signaler la nouvelle situation. Coté client, au niveau de la couche de communication de groupe, lorsqu'un délai de garde signale que le groupe contacté ne répond plus, une nouvelle recherche permet de mettre à jour l'association *identifiant de la donnée/localisation du LDG*. En effet, au gré des occurrences de fautes et des créations de nouvelles copies, les groupes sont susceptibles de se

[5]De même que pour les identifiants de nœuds et de GDG LDG, l'identifiant de la donnée est générée au niveau JXTA et est considéré comme étant unique.

"déplacer" sur les nœuds physiques. Ainsi les fautes sont invisibles au niveau des couches supérieures.

Gestion des fausses suspicions. Sans faire d'hypothèse de synchronisme, il n'est jamais possible d'être sûr qu'un nœud a subit une faute : il est peut-être simplement très lent [49]. Aussi les informations remontées par les détecteurs de fautes ne sont que des suspicions. Il est possible qu'un nœud soit suspecté d'être fautif alors qu'il ne l'est pas. C'est ce que l'on appelle, une fausse suspicion. Dans les cas où un fournisseur membre d'un groupe est suspecté, tous les groupes auxquels il appartient (un groupe par donnée stockée) pour lesquels au moins un des membres va le suspecter[6] vont le considérer comme fautif et l'éliminer des listes de membres et éventuellement le remplacer. S'il s'agit d'une fausse suspicion, cela signifie que le nœud suspect continue de s'exécuter et ceci indépendamment du reste du groupe dont il a été évincé. Une telle situation peut mener à une incohérence : pour chaque donnée stockée par ce fournisseur suspect, l'état du protocole de cohérence ainsi que la donnée stockée, n'évoluent plus comme le reste du groupe d'origine. Pour remédier à cela deux solutions sont possibles : la réparation ou le "suicide". La réparation consiste à réintégrer le fournisseur faussement suspecté au sein du groupe. C'est une solution compliquée et coûteuse : le fournisseur peut appartenir à de nombreux groupes et qui plus est de nouvelles suspicions peuvent survenir lors du processus de réparation. Il est donc plus simple et moins coûteux d'utiliser le suicide, c'est-à-dire considérer le nœud faussement suspecté comme réellement fautif, car les différents groupes suspectant le fournisseur ont déjà déclenché le processus d'adaptation aux fautes. En revanche, le suicide ne concerne que les architectures en couches instanciées pour les données pour lesquelles le fournisseur a été suspecté. En effet, il est possible que le fournisseur participe encore activement à des groupes par lesquels il n'a pas été suspecté. Un avantage de cette politique est que si le nœud sur lequel s'exécute le fournisseur a été suspecté, c'est peut-être parce qu'il est surchargé. Le suicide pour certains groupes va alors permettre d'alléger la charge du nœud suspect.

Défaillances des clients. Nous ne traitons pas ici des défaillances des clients. En effet si un client JUXMEM est défaillant, le processus client applicatif associé l'est également : ils s'exécutent sur le même nœud et nous considérons les défaillances de type *panne franche* (voir section 3.1.1). Les politiques de réaction aux défaillances des clients vont dépendre des mécanismes de tolérance aux fautes de l'application. L'application peut-elle continuer son exécution alors qu'un processus s'arrête ? Si c'est le cas, le protocole de cohérence doit évincer ce client des utilisateurs potentiels de la donnée et si nécessaire régénérer l'objet de synchronisation détenu par ce client. Cependant, que se passe-t-il en cas de fausse suspicion d'un client si un objet de synchronisation a été redistribué à tort ? La réponse à ces questions est difficile, elle dépend notamment du degré d'interaction entre le service de partage de données et les applications.

Supporter les défaillances des clients nécessite une interaction très poussée entre le service de partage de données et les applications. De plus, cela exige que les applications elles-mêmes supportent les fautes car les clients du service de partage de données sont les processus de l'application distribuée. Il existe de nombreux mécanismes qui permettent aux

[6] Dans la pratique, les communications intragrappe des détecteurs de fautes étant basées sur IP-multicast, un nœud suspecté par un membre le sera "très vite" par tous les membres.

applications distribuées de tolérer les fautes, ils sont détaillées dans la section 3.4. Cependant ils sont généralement conçus soit pour des grappes de calculateurs et ne passent pas à l'échelle [47], soit pour une très grande échelle (Internet) et ne tirent pas avantage de la topologie hiérarchique des grilles de calcul [22]. Dans la section suivante nous présentons un mécanisme de sauvegarde de points de reprise pour applications parallèles hiérarchiques (comme les applications de couplage de code), adapté aux fédérations de grappes et en particuliers à notre service de partage de données JUXMEM. Un tel mécanisme permet la gestion des défaillances des clients.

7.4 Mécanismes complémentaires

Les défaillances des clients ne peuvent pas être prises en compte par le service de partage de données sans mécanismes complémentaires. En effet, les processus de l'application distribuée s'exécutent sur les nœuds clients, et en cas de faute, le processus fautif peut être relancé à partir d'un état précédemment sauvegardé. Mais, il existe des dépendances entre ces processus. Lorsqu'un processus retourne en arrière dans un état sauvegardé, il est nécessaire de prendre en compte ces dépendances afin d'éviter les incohérences qui peuvent être introduites par les messages qu'il va ré-émettre, ou ceux qu'il avait déjà reçus et qu'il risque d'attendre indéfiniment. La solution consiste à sauvegarder un état global cohérent, un *point de reprise*, c'est-à-dire un ensemble composé des états locaux de tous les processus applicatifs tels qu'il n'existe pas de dépendance entre ces états. La section 3.4 présente plusieurs travaux de recherche sur la sauvegarde d'états globaux cohérents.

Le service de partage de données peut servir à sauvegarder les points de reprise. Il doit collaborer avec le mécanisme de sauvegarde de point de reprise afin que, en cas de retour en arrière de l'application distribuée, les données partagées et les états des protocoles de cohérence puissent être eux aussi restaurés pour éviter une incohérence.

Cette coopération permet de mettre en place au niveau du service de partage de données des mécanismes de réplication très optimistes : en cas de perte de la dernière version de la donnée, il devient possible d'effectuer un retour arrière dans un état cohérent pour lequel il existe une sauvegarde de la donnée perdue.

Des mécanismes de sauvegarde de points de reprises hiérarchiques. Malgré le grand nombre de mécanismes de points de reprises, il en existe très peu qui soient adaptés à la grille. En effet, ils sont en général conçus soit pour des petites échelles et impliquent des synchronisations globales qui ne sont pas adaptées aux grilles de calcul, soit pour des très grandes échelles et ne permettent pas de tirer avantage des réseaux haute performance présents au sein des grappes.

Nous avons donc conçu un protocole de points de reprise pour les applications parallèles s'exécutant sur des grilles de calcul. Ce protocole utilise une technique *hiérarchique hybride*, mettant en place des sauvegardes de points de reprise *coordonnées* au sein des grappes, *induites par les communications* entre les grappes. Au niveau intergrappe, les grappes sont vues comme des "super-nœuds" par le protocole de sauvegarde de points de reprise induit par les communications. Cela est rendu possible par le protocole de sauvegarde de points de reprise intragrappe qui sauve des points de reprises cohérents grâce à une synchronisation des processus s'exécutant au sein de la grappe. La suite de cette section détaille ce mécanisme.

Points de reprise coordonnés au sein des grappes. Au sein d'une grappe de calculateurs, un mécanisme de sauvegarde de points de reprise coordonnés semble être raisonnable. En effet le surcoût induit par une synchronisation est relativement faible grâce aux performances des réseaux de type SAN (*System Area Network*, réseau à faible latence et débit élevé). Périodiquement, un des nœuds de la grappe devient l'initiateur d'un point de reprise coordonné intragrappe. Il envoie à tous les nœuds de la grappe une requête d'établissement de point de reprise. Lorsqu'un nœud reçoit une telle requête il sauvegarde son état local. Une fois que l'état local est sauvegardé sur un espace de stockage stable (par exemple le service de partage de données pour la grille JUXMEM) il envoie un acquittement à l'émetteur. Lorsque l'initiateur a reçu un acquittement de la part de tous les autres nœuds de sa grappe, il valide son propre état local et envoie un message de validation à tous les autres nœuds afin qu'ils fassent de même. Il est possible que deux nœuds décident de déclencher la sauvegarde d'un point de reprise coordonné de manière concurrente. Dans ce cas, l'identifiant unique du nœud est utilisé pour déterminer lequel est considéré comme l'initiateur. Ces points de reprise sont estampillés et sauvegardés en mémoire stable.

Points de reprise induits par les communications entre les grappes. Les latences intergrappe pouvant être élevées, un mécanisme de points de reprise coordonné impliquant une synchronisation globale sur la grille n'est pas satisfaisant. Les points de reprise coordonnés décrits au paragraphe précédent sont sauvegardés pour chaque grappe de manière indépendante. Nous avons vu à la section 3.4 que dans le cas de points de reprise indépendant, il pouvait y avoir un *effet domino* (important retour en arrière des processus à la recherche d'un état global cohérent). L'effet domino peut être éliminé en *forçant* la sauvegarde de certains points de reprise lors des communications en fonction d'informations ajoutées aux messages. Ce sont en effet les communications qui impliquent des dépendances : dans un état global cohérent, tout message reçu doit avoir été émis. Lors d'une communication intergrappe, la grappe dans laquelle se situe le nœud récepteur peut être conduite à la sauvegarde d'un point de reprise coordonné. Les techniques de points de reprise induits par les communications ont été classifiées dans [80]. Cette classification tient essentiellement compte du nombre de points de reprise forcés "inutiles". Forcer un point de reprise avant chaque réception de message, par exemple, empêche bien l'effet domino mais ajoute un surcoût important et inutile. Nous nous intéressons donc aux conditions permettant de prendre un nombre minimum de points de reprise forcés. Nous proposons d'utiliser un vecteur de dépendance directe (*Direct Dependancy Vector* ou DDV) comme décrit dans [8] afin de détecter les dépendances entre les grappes et de forcer la sauvegarde de points de reprise de manière appropriée. Nous proposons également de journaliser les messages en transit. La gestion des DDV implique que chaque grappe gère un numéro de séquence pour estampiller ses points de reprise, ce numéro est incrémenté chaque fois qu'un point de reprise est établi. Le DDV d'une grappe est un vecteur dont chaque entrée est le dernier numéro de séquence de la grappe voisine correspondante. Ce numéro de séquence (noté SN) est ajouté à chaque message intergrappe. Lors de la réception d'un message provenant d'une autre grappe, le numéro de séquence qu'il contient est comparé avec l'entrée du DDV correspondante. S'il est supérieur, un point de reprise est forcé avant de prendre le message en compte.

Retour en arrière des applications. Quand un nœud de type client dans une grappe est fautif (panne franche), les nœuds clients de la même application s'exécutant dans la même

grappe effectuent un retour arrière et alertent les nœuds clients des autres grappes. Lorsqu'un nœud client reçoit une telle alerte (contenant le numéro de séquence du point de reprise restauré sur le site émetteur) le DDV est utilisé afin de calculer s'il est nécessaire d'effectuer un retour en arrière dans la grappe réceptrice : c'est le cas si et seulement si elle possède un point de reprise pour lequel l'entrée du DDV correspondant au site ayant envoyé l'alerte est supérieure ou égale au numéro de séquence reçu.

Un tel mécanisme, ainsi que sa coopération avec le service de partage de données, est complexe a mettre en œuvre, mais c'est à ce prix que les défaillances des clients peuvent être prisent en compte.

7.5 La hiérarchie : une solution générique pour les grilles ?

Dans ce chapitre nous avons montré comment de nombreux protocoles de cohérence peuvent être adaptés aux grilles en les concevant de manière *hiérarchique*. Nous avons également décrit notre *hiérarchie* de groupes auto-organisants permettant de mettre en place des mécanismes de réplication adaptés aux grilles. Enfin nous avons étudié comment il est possible de mettre en place des mécanismes de sauvegarde de points de reprise pour les applications distribuées s'exécutant sur les grilles en employant une technique *hiérarchique*.

Le point commun entre ces différentes solutions réside dans leur conception *hiérarchique*. Pour chacune de ses solutions, la motivation d'une conception hiérarchique est la même : limiter l'utilisation des liens intergrappe à forte latence et tirer profit des réseaux haute performance au sein des grappes. En effet, si l'on considère les grilles comme des fédérations de grappes, la topologie du réseau d'interconnexion (décrite dans la section 2.1) est clairement *hiérarchique*.

Limites de l'approche hiérarchique. Bien qu'il semble que les approches hiérarchiques soient bien adaptées aux grilles, elles présentent certains inconvénients.

En premier lieu, toutes les grilles de calcul ne sont pas des fédérations de grappes, comme expliqué dans la section 2.1 et leur topologie n'est alors pas aussi simple. Il est alors possible de généraliser notre solution en utilisant des approches mettant en place des hiérarchies à plus de deux niveaux. De telles solutions seraient cependant très complexes.

En deuxième lieu, la mise en place de solutions *hiérarchiques* s'avère complexe. Il est nécessaire de 1) détecter la hiérarchie physique, 2) faire correspondre la hiérarchie logicielle avec la hiérarchie physique, et surtout 3) maintenir les mécanismes hiérarchiques en présence de fautes.

Enfin, une approche hiérarchique implique souvent un plus grand nombre de sauts dans le réseau. Sur l'exemple du protocole de cohérence hiérarchique du chapitre 5 on observe en effet que dans certains cas le client communique avec la copie de référence locale qui contacte la copie de référence globale avant de répondre au client. Dans le cas de la version non hiérarchique les communications sont restreintes à celles entre le client et la copie de référence globale. Il apparaît donc qu'une approche hiérarchique réduit la durée moyenne des sauts (en favorisant l'utilisation des liens à faible latence) mais peut également en augmenter le nombre.

Troisième partie

Mise en œuvre et évaluation

Chapitre 8

Exemple de mise en œuvre d'un protocole de cohérence tolérant aux fautes

Sommaire

L'architecture en couches permettant une gestion conjointe de la tolérance aux fautes et de la cohérence des données présentée dans le chapitre 6 a été mise en œuvre au sein du service de partage de données JUXMEM [129]. Une mise en œuvre de protocoles de cohérence hiérarchiques tolérants aux fautes suivant le modèle décrit au chapitre 7 a également été réalisée au sein de cette architecture.

Deux thèses (celle de Mathieu Jan et celle-ci) ainsi que trois stages de master recherche deuxième année ont contribué au développement du service de partage de données JUXMEM. Le prototype mettant en œuvre JUXMEM correspond à plus de 16 700 lignes de code Java et plus de 13 500 lignes de code C, dont environ 7 300 lignes de code Java et 6 100 lignes de code C pour la gestion de la cohérence des données et de la tolérance aux fautes.

Dans ce chapitre, nous nous focalisons sur la mise en œuvre de notre contribution. La section 8.1 décrit la mise en œuvre de l'architecture en couches présentée dans le chapitre 6 et les sections 8.2 et 8.3 présentent respectivement un exemple de mise en œuvre d'un protocole de cohérence et un exemple de mise en œuvre de la couche de tolérance aux fautes au sein de cette architecture. Enfin dans la section 8.4, nous nous intéressons à une extension du protocole de cohérence présenté dans la section 8.2 offrant aux applications la possibilité d'effectuer des observations efficaces d'une donnée partagée.

8.1 Mise en œuvre de l'architecture en couches

Le rôle de l'architecture en couches est de découpler la gestion de la tolérance aux fautes de la gestion de la cohérence des données. Cela permet d'une part de pouvoir se concentrer sur un problème unique lors de la mise en œuvre des différentes couches et d'autre part d'offrir différentes implémentations de chacune des couches. Grâce à ce dernier point, notre prototype peut offrir aux applications le choix parmi un éventail de combinaisons.

8.1.1 Architecture logicielle générale

Pour clarifier notre présentation, nous considérons que chaque nœud physique héberge un seul pair JUXMEM. Nous considérons également que chaque pair JUXMEM ne joue qu'un seul rôle au sein de la plate-forme, c'est-à-dire que chaque pair est soit client, soit fournisseur, soit gestionnaire[1]. Ces différents rôles sont décrits en détail dans la section 6.1. Les pairs clients sont liés aux processus applicatifs et permettent d'accéder aux données partagées présentes au sein du service JUXMEM. Les pairs fournisseurs sont quant à eux responsables du stockage des copies des données partagées. Toujours à des fins de clarté, nous nous focalisons ici sur une seule donnée, même si les clients peuvent accéder à plusieurs données partagées, et que les fournisseurs peuvent stocker des copies de plusieurs données différentes.

Instantiation des couches sur les pairs JUXMEM. Pour chaque donnée partagée à laquelle il accède, un client doit instancier la partie client du protocole de cohérence choisi ainsi qu'une couche de tolérance aux fautes, c'est-à-dire la pile logicielle représentée par la figure 8.1. Le rôle de chacune des couches de cette pile logicielle est décrit au chapitre 6. Le protocole de cohérence utilise la couche de tolérance aux fautes via la couche de jonction afin de communiquer avec la copie de référence répliquée sur des pairs fournisseurs.

Chaque donnée stockée au sein du service JUXMEM est répliquée, c'est-à-dire qu'elle est présente en plusieurs exemplaires. Nous avons décrit au chapitre 7 notre architecture hiérarchique pour la réplication et la gestion de la cohérence. Chaque donnée fait donc intervenir

[1] Notre implémentation permet cependant d'exécuter plusieurs pairs sur un même nœud physique, et chaque pair peut jouer chacun des rôles, simultanément, ou à tour de rôle.

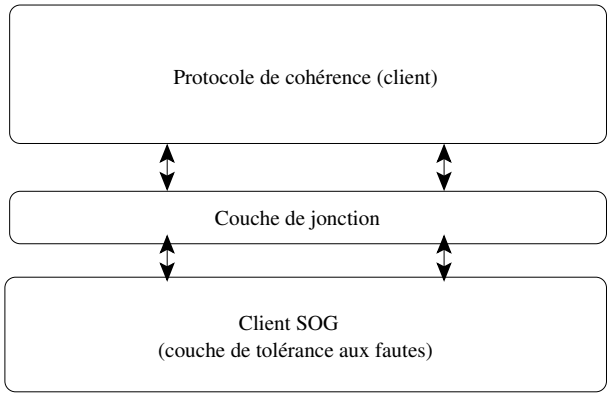

FIG. 8.1 – Pile logicielle instanciée sur les pairs clients pour chaque donnée partagée à laquelle le client accède.

plusieurs pairs fournisseurs. Les pair fournisseurs hébergeant des copies d'une même donnée et appartenant à une même grappe (groupe *cluster* JUXMEM) forment un *Local Data Group* (LDG) qui est responsable d'une copie de référence locale. L'ensemble des LDG forme le *Global Data Group* (GDG) qui est responsable de la copie de référence globale (voir chapitre 7). Le LDG est un groupe dont les membres sont eux-mêmes des groupes, c'est donc un groupe de groupes. Chaque pair fournisseur hébergeant une copie d'une donnée appartient à un LDG *et* chaque LDG appartient à un GDG. Pour chaque donnée, les deux piles logicielles représentées par la figure 8.2 doivent être instanciées sur *tous* les pairs fournisseurs qui en hébergent une copie. Un exemple de groupe hiérarchique de fournisseurs formant un GDG comprenant deux LDG est illustré par la figure 8.3.

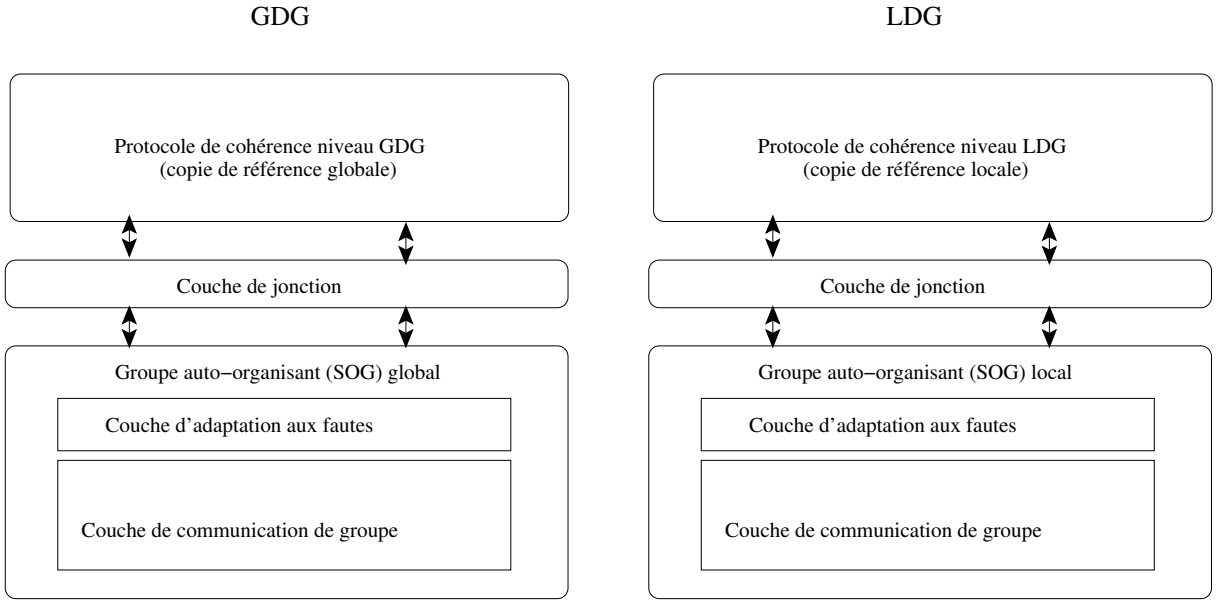

FIG. 8.2 – Piles logicielle instanciée sur les pairs fournisseurs pour chaque donnée qu'ils hébergent.

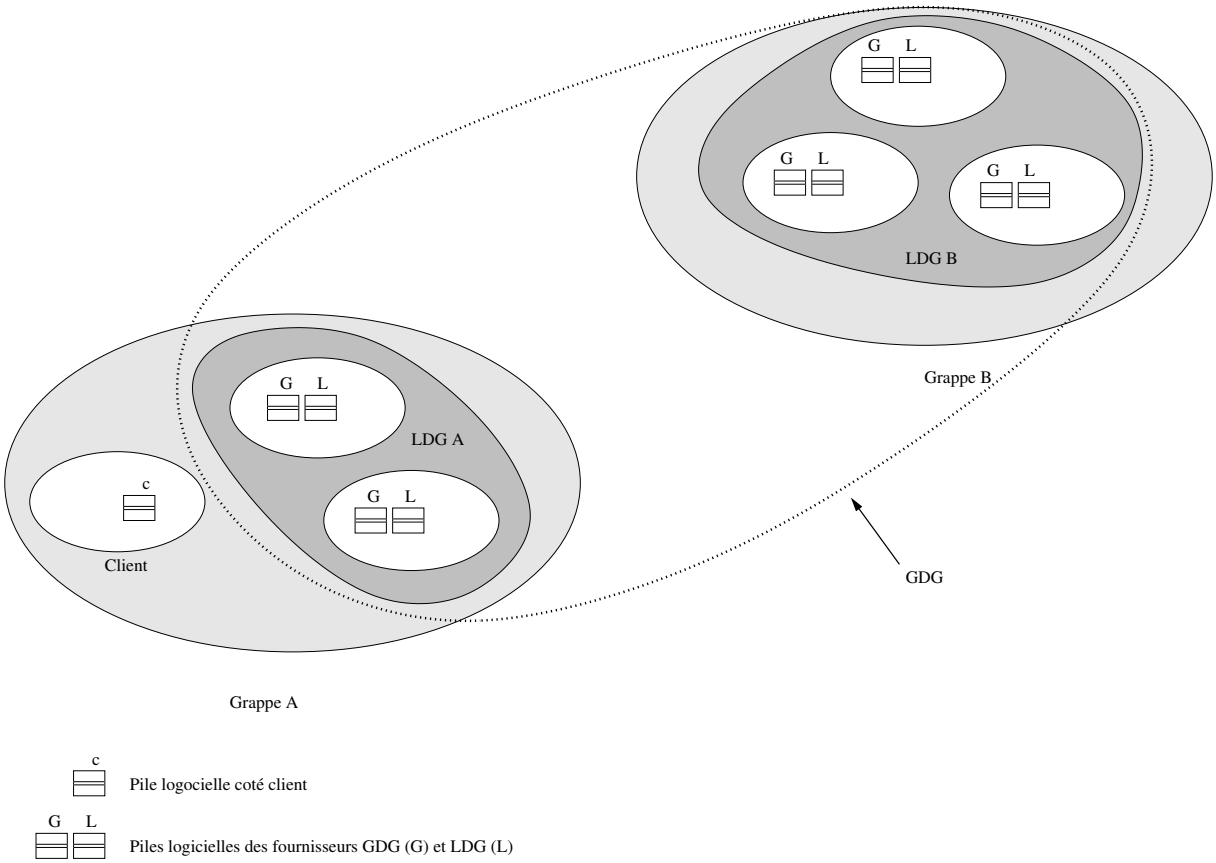

FIG. 8.3 – Aperçu de l'architecture logicielle : un client et un GDG composé de deux LDG eux-mêmes composés de 2 à 3 pairs fournisseurs.

8.1.2 Les protocoles de cohérence

Les processus applicatifs utilisent l'interface de JUXMEM lors des accès aux données partagées. Cette dernière utilise alors l'interface offerte par les protocoles de cohérence[2], représentée par le listing 8.1.

Listing 8.1 – Interface des protocoles de cohérence des clients JUXMEM.

```
1  ...
2  /* l'objet peer est éutilis pour éaccder aux fonctions du noyau JuxMem */
3  Jxta_id* cp_alloc(JuxMem_peer* peer, Jxta_vector* local_provider_ids,
4                    Jxta_vector* remote_provider_ids,
5                    Jxta_id** provider_comm_id, long size,
6                    int cp_type, int sog_type);
7  consistency_protocol* consistency_protocol_open(JuxMem_peer* peer,
8                                                  Jxta_id* data_id,
9                                                  Jxta_id* provider_comm_id,
10                                                 long size, char* localkey,
11                                                 void* attributes);
12 /* self éreprsent le protocole de écohrence */
```

[2]Pour une donnée particulière, un seul protocole de cohérence est employé. Cependant, pour un même client, des protocoles de cohérence différents peuvent être utilisés pour des données différentes.

```
13 void consistency_protocol_close(consistency_protocol* self);
14 int cp_flush(consistency_protocol* self, void* ptr, size_t size,
15              Jxta_message_element* msg_element_localkey);
16 int cp_acquire(consistency_protocol* self, void* ptr, size_t size,
17               Jxta_message_element* msg_element_localkey);
18 int cp_release(consistency_protocol* self, void* ptr, size_t size,
19               Jxta_message_element* msg_element_localkey);
20 int cp_acquire_read(consistency_protocol* self, void* ptr, size_t size,
21                     Jxta_message_element* msg_element_localkey);
22 int cp_get_state(consistency_protocol* self);
23 ...
```

Les protocoles de cohérence doivent donc définir ces différentes primitives. Les primitives cp_acquire et cp_acquire_read sont appelées avant chaque entrée dans une section de code contenant des accès à la donnée : cp_acquire lorsque les accès sont en lecture/écriture et cp_acquire_read lorsqu'ils sont en lecture seule. La primitive cp_release est appelée à la sortie des sections de code contenant des accès à la donnée. Les autres primitives sont utilisées par l'interface de JUXMEM pour propager les écritures à la copie de référence locale (cp_flush) ou pour connaître l'état du protocole de cohérence (cp_get_state). Pour mettre en œuvre ces primitives, les protocoles de cohérence utilisent les fonctions de l'interface de la couche de jonction dont un extrait est présenté dans le listing 8.2.

Listing 8.2 – Interface de communication offerte par la couche de jonction.

```
1  ...
2  /* self éreprsente ici la couche de jonction */
3  int send_lock_request(client_grp_comm* self, Jxta_message* message);
4  int send_lock_read_request(client_grp_comm* self, Jxta_message* message);
5  int send_unlock_request(client_grp_comm* self, Jxta_message* message);
6  int send_unlock_read_request(client_grp_comm* self, Jxta_message* message);
7  int send_read_request(client_grp_comm* self, Jxta_message* message);
8  int send_update_request(client_grp_comm* self, Jxta_message* message);
9  int send_reset_ack(client_grp_comm* self, Jxta_message* message);
10 int set_cp_handler(client_grp_comm* self, consistency_protocol* cp);
11 ...
```

Il est intéressant de remarquer que ces fonctions ne prennent pas en paramètre l'identifiant du destinataire du message. En effet, le destinataire est un groupe de copies dont les membres sont susceptibles de changer au cours du temps, lors d'occurrences de fautes. L'identifiant du groupe est inconnu au niveau de cette couche. La localisation du groupe et les communications avec ce groupe sont donc à la charge de la couche de tolérance aux fautes dont un exemple de mise en œuvre est décrit à la section 8.3. Il n'est par conséquent pas nécessaire de connaître ni la localisation, ni la composition du LDG/GDG au niveau du protocole de cohérence.

Le protocole de cohérence client doit également définir la manière dont il réagit aux réponses du LDG. Pour ce faire, il doit mettre en œuvre les fonctions du listing 8.3 appelées par la couche de jonction lors de la réception d'un message de ce dernier.

Listing 8.3 – Fonctions du protocole de cohérence client appelées lors de la réception d'un message de la copie de référence locale.

```
1 ...
2 /* self éreprsente ici le protocole de écohrence */
3 void recv_lock_ack(consistency_protocol* self, Jxta_id* sender,
```

```
4                       Jxta_message* message);
5   void recv_lock_read_ack(consistency_protocol* self, Jxta_id* sender,
6                           Jxta_message* message);
7   void recv_unlock_ack(consistency_protocol* self, Jxta_id* sender,
8                        Jxta_message* message);
9   void recv_unlock_read_ack(consistency_protocol* self, Jxta_id* sender,
10                            Jxta_message* message);
11  void recv_update_ack(consistency_protocol* self, Jxta_id* sender,
12                       Jxta_message* message);
13  void recv_read_ack(consistency_protocol* self, Jxta_id* sender,
14                     Jxta_message* message);
15  void recv_reset_request(consistency_protocol* self, Jxta_id* sender,
16                          Jxta_message* message);
17  ...
```

La partie des protocoles de cohérence mise en œuvre au niveau de la copie de référence locale, sur les pairs fournisseurs doit implémenter les fonctions miroirs (recv_$*$ pour chacun des send_$*$) qui seront appelées par la couche de jonction. Elle utilise de plus une interface semblable à celle du listing 8.2 afin d'émettre des requêtes à la copie de référence globale.

De la même manière, la copie de référence globale doit implémenter les fonctions recv_$*$ pour chacune des requêtes émises par les copies de référence locales. À ce niveau également, la couche de jonction propose des fonctions du type send_$*$ permettant à la copie de référence globale de répondre aux requêtes des copies de référence locales.

Au niveau de l'implémentation d'un protocole de cohérence, la tolérance aux fautes est rendue invisible grâce à ces interfaces. En effet, le protocole de cohérence utilise les fonctions send_$*$ et recv_$*$ de la couche de jonction, masquant la localisation et la composition des groupes pour les communications entre les clients, les copies de références locales et la copie de référence globale. Ces trois entités d'un protocole de cohérence implémentent le protocole de cohérence hiérarchique en mettant en œuvre des automates à états semblables à ceux présentés dans le chapitre 5. Un exemple de mise en œuvre est donnée à la section 8.2.

8.1.3 Les mécanismes de tolérance aux fautes

La couche de tolérance aux fautes est responsable de la mise en œuvre des groupes auto-organisants (SOG, pour l'anglais *Self-Organizing Group*). La partie instanciée du côté client est responsable de la localisation et de la communication avec le LDG présent dans la même grappe, voire de l'instanciation d'un nouveau LDG s'il n'en existe pas au sein de la grappe du client. C'est à ce niveau que l'identifiant du groupe est conservé. La partie instanciée au niveau LDG des pairs fournisseurs est responsable de la réplication au sein des LDG, de l'auto-organisation des LDG, ainsi que de la localisation et des communications avec le GDG. Enfin, la partie instanciée au niveau GDG des pairs fournisseurs est responsable de la réplication au niveau global (c'est-à-dire interLDG), ainsi que de l'auto-organisation du GDG qui peut entraîner l'ajout et/ou le retrait de LDG.

Localisation et communication. L'interface de communication offerte à la couche de jonction est simple : elle permet à cette dernière d'envoyer des messages à un groupe et d'enregistrer une fonction permettant de recevoir et de traiter les messages envoyés au groupe. Ces fonctions sont données dans le listing ci-dessous.

Listing 8.4 – Interface de communication offerte par la couche de tolérance aux fautes à la couche de jonction.

```
1 ...
2 int grp_send(sog_client_stub* self, Jxta_message* message);
3 int set_msg_evt_handler(sog_client_stub* self, client_grp_comm* handler);
4 ...
```

En plus de ces fonctions, des *étiquettes* ajoutées aux messages permettent à la couche de jonction d'influer sur le comportement de la couche de tolérance aux fautes. Par exemple, si l'étiquette GRP_COMM est ajoutée par la couche de jonction, le message sera distribué aux membres du groupe en utilisant un mécanisme de diffusion atomique. Cette étiquette est donc ajoutée à chacun des messages modifiant l'état du protocole de cohérence (requête de verrouillage, de déverrouillage ou de mise à jour de la donnée). Ceci assure que tous les membres d'un même groupe appliqueront les *mêmes mises à jour*, et ce dans le *même ordre*. En revanche, en l'absence de cette étiquette, le message ne sera transmis qu'à l'un des membres du groupe. Une requête de lecture de la donnée d'un client à son LDG n'engendrera ainsi pas de communication de groupe, mais seulement une communication entre le client et un des pairs fournisseurs appartenant au LDG associé présent dans la même grappe. L'ajout ou non de cette étiquette se décide au niveau de la couche de jonction afin de préserver la transparence de la réplication au niveau du protocole de cohérence.

Les groupes auto-organisants utilisent les interfaces de publication/recherche et de communication du noyau JUXMEM présentée à la section 6.1.3. L'interface de publication/recherche permet de localiser les groupes grâce à l'identifiant de la donnée : les clients doivent pouvoir localiser le LDG présent au sein de leur grappe et les LDG doivent pouvoir localiser le GDG. L'interface de communication du noyau permet au SOG d'enregistrer une fonction qui sera appelée chaque fois qu'un message concernant la donnée pour laquelle il a été instancié est reçu.

Auto-organisation des groupes. Les SOG s'enregistrent également auprès du service hiérarchique de détection de fautes dont l'intégration au service de partage de données JUXMEM est présentée à la section 7.2.1. En cas de faute détectée, la fonction `int repair(Jxta_id* faulty_peer_id)` est appelée. Cette fonction a pour rôle : 1) d'évincer le fournisseur fautif des listes de membres auxquelles il appartient ; 2) de décider si le fournisseur doit être remplacé ou non ; 3) de le remplacer si cela est nécessaire ; et 4) de mettre à jour les informations publiées afin que le groupe reste localisable par l'intermédiaire de l'interface de publication/recherche du noyau.

8.2 Mise en œuvre d'un protocole de cohérence hiérarchique

Pour mettre en œuvre un protocole de cohérence hiérarchique au sein de notre architecture en couche, il est nécessaire de mettre en œuvre différentes entités : 1) la partie client, 2) la copie de référence locale, et 3) la copie de référence globale. Nous reprenons ici l'exemple du chapitre 5 implémentant le modèle de cohérence à l'entrée (décrit au chapitre 4).

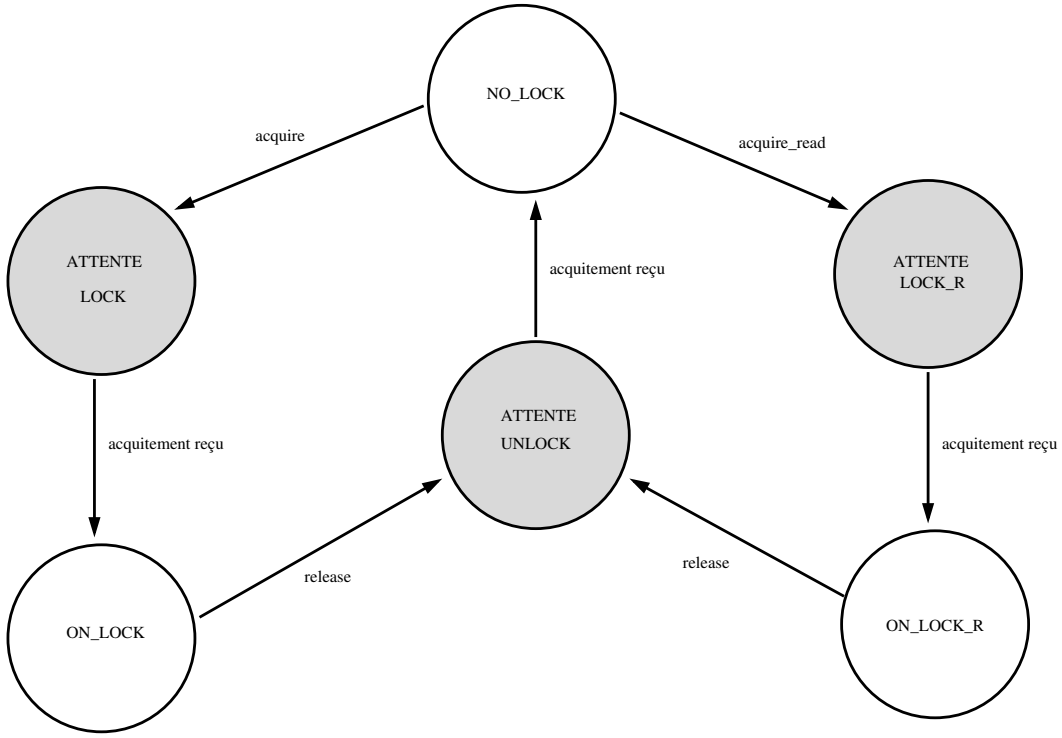

FIG. 8.4 – Automate à états représentant le protocole de cohérence à l'entrée au niveau du client. Les états non grisés ne bloquent pas l'application alors que les états grisés correspondent à des primitives synchrones.

8.2.1 Mise en œuvre sur le client

La partie client du protocole de cohérence doit mettre en œuvre à la fois l'interface présentée à JUXMEM, illustrée par le listing 8.1, et l'interface susceptible d'être appelée par la couche de jonction présentée sur le listing 8.3.

Un exemple de mise en œuvre consiste à conserver un état (possession ou non du verrou en écriture/en lecture, témoin de modification de la donnée, version de la donnée présente localement, etc.) et, lorsque l'une des fonctions est appelée, effectuer une *action* puis mettre l'état à jour. L'action dépend de l'état ainsi que de la fonction appelée. Cela permet de mettre en place un automate à états (figure 8.4).

Certaines actions (ou transitions de l'automate à états) nécessitent des communications avec la copie de référence locale. Celles-ci sont réalisées via l'interface de communication de la couche de jonction présentée par le listing 8.2. Le listing 8.5 ci-dessous montre un exemple d'implémentation de la fonction appelée lorsqu'un client souhaite acquérir le verrou permettant un accès exclusif à la donnée.

Listing 8.5 – Code simplifié de l'acquisition d'un verrou exclusif par le protocole de cohérence client.

```
1 int ec_acquire(consistency_protocol* self, void *ptr, size_t size,
2                Jxta_message_element* msg_element_localkey)
3 {
```

```
4       ...
5       if (self->impl->state == NO_LOCK) {
6           msg = jxta_message_new();
7           /* le énumro de version est éajout au message */
8           status = consistency_version_to_msg(self->impl->data_version, msg);
9           /* la êrequte est éenvoye à la couche de jonction */
10          send_lock_request(self->impl->comm, msg);
11          ...
12          /* un acquitement est attendu */
13          jxta_listener_wait_for_event(self->impl->lock_listener,
14                                       TIME_TO_WAIT, JXTA_OBJECT_PPTR(&obj));
15          ...
16          /* la édonne contenue dans l'acquitement est éércupre */
17          juxmem_to_user_memory_from_message(self->impl->peer,
18                                             self->impl->data_id,
19                                             ptr, size, obj);
20          ...
21          /* lé'tat est mis à jour */
22          self->impl->state = ON_LOCK;
23      }
24      ...
25  }
```

Si le protocole est dans l'état NO_LOCK (ligne 5), une requête d'obtention de verrou est émise au LDG (ligne 10) via la couche de jonction, puis un acquittement du LDG est attendu (lignes 13 et 14) avant de mettre la donnée et l'état à jour (lignes 17 et 22). Les autres transitions sont implémentées selon le même principe.

L'automate à états représenté par la figure 8.4 montre que, lors de la mise en œuvre du protocole de cohérence à l'entrée sur le client, il n'est pas nécessaire d'avoir connaissance ni de la réplication, ni de la mise en œuvre hiérarchique du protocole de cohérence. Nous avons remarqué à la section précédente que la couche jonction rendait la réplication transparente au niveau du protocole de cohérence. Cela permet notamment de pouvoir remplacer facilement la couche de tolérance aux fautes et de conserver le protocole de cohérence intact. La hiérarchie est également transparente pour le protocole de cohérence sur le client : tout se passe comme si la copie de référence locale à laquelle il accède était la seule et unique copie de référence.

8.2.2 Mise en œuvre sur les fournisseurs

Nous avons mentionné à la section 8.1 que notre implémentation hiérarchique exige que chaque fournisseur possédant une copie d'une donnée instancie à la fois la partie du protocole de cohérence implémentant la copie de référence locale et celle implémentant la copie de référence globale.

Copie de référence locale. La mise en œuvre d'une copie de référence locale nécessite l'implémentation des fonctions susceptibles d'être appelées par la partie communication de la couche de jonction, c'est-à-dire les fonctions correspondant à la réception de messages d'un client ou de la copie de référence globale.

Une copie de référence locale se comporte comme un automate à états, semblable à celui représentant le comportement de la copie de référence du protocole de cohérence à l'entrée

présenté au chapitre 5. De même que pour la partie client du protocole de cohérence, sa réalisation peut être effectuée en maintenant un état et en effectuant des actions chaque fois qu'un message provenant de la copie de cohérence globale ou d'un client est reçu. Dans le cas de la copie de référence locale, l'état de l'automate est cependant plus complexe, il contient notamment l'identifiant du client possédant le verrou, et les listes suivantes : 1) une liste de clients en attente du verrou exclusif (L pour *Lock*) ; 2) une liste de clients en attente du verrou en lecture (LR pour *Lock Read*) ; 3) une liste de clients possédant le verrou en lecture (OLR pour *On Lock Read*).

L'interface de communication de la couche de jonction permet à la copie de référence locale d'envoyer des messages aux clients (par exemple des acquittements de requêtes de demandes de verrous) ou à la copie de référence globale (par exemple une demande de verrou).

Copie de référence globale. La mise en œuvre d'une copie de référence globale s'effectue en implémentant les fonctions susceptibles d'être appelées par la couche de jonction. Elles correspondent à l'arrivée de différents types de message émis par une copie de référence locale.

Dans notre mise en œuvre du protocole de cohérence à l'entrée, la copie de référence globale se comporte comme une copie de référence locale à deux différences près : 1) la copie de référence globale communique avec des copies de référence locales, et non avec des clients ; et 2) la copie de référence globale étant au dernier niveau de la hiérarchie, elle ne communique pas avec un niveau supérieur.

8.2.3 Fonctionnement

La figure 8.5 illustre le fonctionnement du protocole de cohérence hiérarchique. Deux clients accèdent à une même donnée partagée via leur copie de référence locale. On remarque que certaines actions ne font intervenir que la partie client du protocole de cohérence (1a). Cela peut être par exemple la lecture d'une donnée présente localement. D'autres actions, comme l'acquisition d'un verrou, font intervenir la copie de référence locale (1b et 2b). Il est possible que la copie de référence locale doive contacter la copie de référence globale pour satisfaire certaines requêtes (1c, 2c, 3c et 4c). Cela peut être le cas lors d'une demande de verrou par le client (1c), si la copie de référence locale ne possède pas ce verrou. Dans ce cas, elle le demande à la copie de référence globale (2c) qui soit le possède, soit attend qu'une autre copie de référence locale le libère. Quand le verrou est en possession de la copie de référence globale, elle le retourne à la copie de référence locale (3c) qui le transmet à son tour au client qui en avait fait la demande (4c).

Pour cet exemple, nous n'avons pas pris en compte la réplication engendrée par les mécanismes de tolérance aux fautes. La copie de référence globale peut se situer sur le même nœud que l'une des copies de référence locales, ou sur un nœud différent au sein d'une des deux grappes ou encore dans une grappe tierce. Avec notre mise en œuvre de groupes hiérarchiques, détaillée à la section suivante, la copie de référence globale est répliquée sur tous les nœuds du système hébergeant une copie de référence locale.

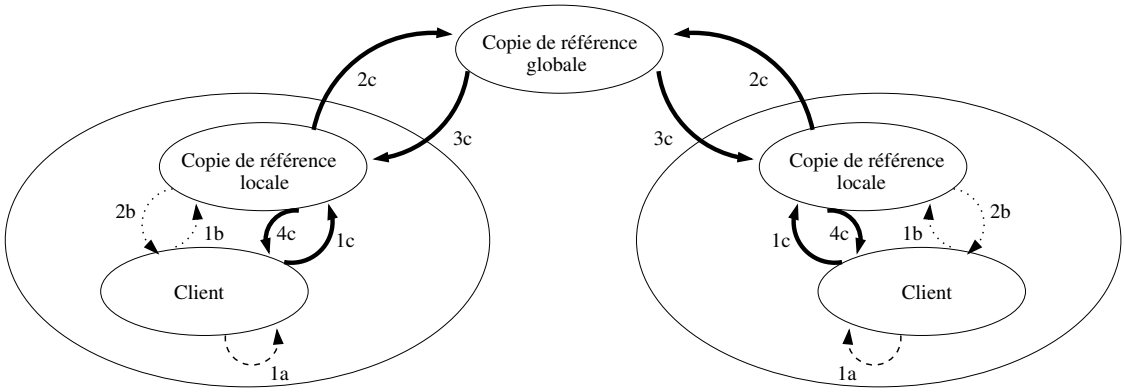

FIG. 8.5 – Fonctionnement du protocole de cohérence hiérarchique.

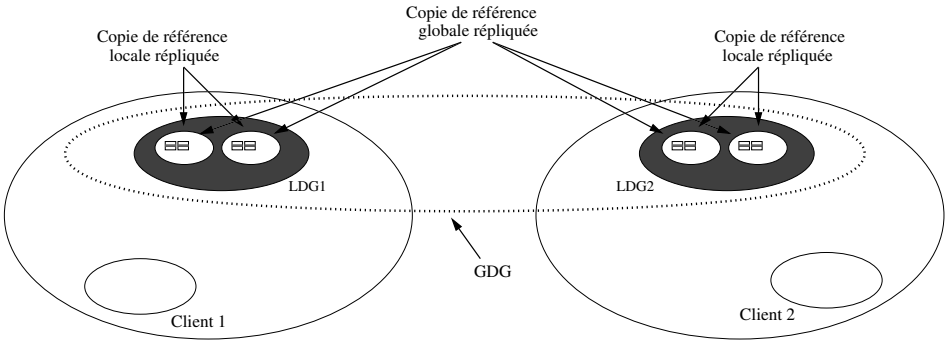

FIG. 8.6 – Correspondance entre les copies de référence locales/globale et des groupes auto-organisants LDG/GDG.

8.3 Mise en œuvre des groupes auto-organisants

Les copies de références locales et globales sont répliquées. Chaque LDG héberge une copie de référence locale répliquée, et chaque GDG héberge une copie de référence globale répliquée. La figure 8.6 illustre cette correspondance. Le rôle des groupes auto-organisants, ou SOG, est d'assurer la persistance des copies de référence en présence de fautes.

8.3.1 Mise en place de la réplication

Les groupes de copies sont initialement créés lors de l'allocation d'une nouvelle donnée au sein de JUXMEM. Les messages envoyés aux groupes sont alors traités de manière à conserver les différentes copies cohérentes.

Allocation. Lorsqu'un processus applicatif souhaite allouer de l'espace pour une donnée au sein de JUXMEM, il doit fournir le degré de réplication souhaité : dans combien de grappes la donnée doit être présente, et, dans chacune de ces grappes, combien il doit y avoir de

copies de la donnée. Il est possible de demander à ce qu'il n'y ait qu'une copie dans la grappe du processus, dans ce cas aucune garantie de tolérance aux fautes n'est offerte.

Le noyau JUXMEM cherche alors autant de pairs fournisseurs que nécessaire. Les mécanismes mettant en œuvre cette recherche sont détaillés dans [64]. Quand les pairs fournisseurs ont été trouvés, avant d'instancier la pile logicielle coté client, des requêtes d'allocation sont envoyées à chacun des fournisseurs sélectionnés. Chaque requête contient la liste des fournisseurs sélectionnés sous forme d'une liste contenant les listes des fournisseurs de chaque groupe. Ces listes sont utilisées pour initialiser les listes de membres des groupes. Dans la mise en œuvre présentée ici, un nœud appelé nœud *sérialisateur* ordonnance les messages pour son groupe. À l'initialisation, pour chaque groupe, le fournisseur apparaissant en premier sur la liste reçue est considéré comme nœud sérialisateur.

Quand les piles logicielles ont été instanciées sur les fournisseurs, ceux-ci retournent des acquittements au client ayant effectué la requête d'allocation. Le client reste bloqué dans l'attente de tous les acquittements[3]. Quand tous les acquittements attendus ont été reçus, la pile logicielle client pour cette donnée est instanciée et le processus applicatif ayant appelé la primitive d'allocation est débloqué.

Communication de groupe. Conserver la cohérence des différentes copies d'un groupe alors que des accès concurrents sont effectués nécessite d'ordonnancer les accès. En effet, si toutes les copies sont modifiées par les mêmes actions, et ce dans le même ordre, elles évolueront toutes de la même manière.

La diffusion atomique des messages définie à la section 3.3.2 garantit que tous les messages envoyés à un groupe seront pris en compte par chacun des membres de ce groupe dans le même ordre. Plusieurs implémentations de la diffusion atomique existent. Au chapitre 5, nous avons décrit un mécanisme de diffusion atomique s'appuyant sur un algorithme de consensus utilisé pour décider de l'ordre de prise en compte des messages. Nous présentons ici une autre mise en œuvre consistant à utiliser un nœud sérialisateur qui a la responsabilité de déterminer cet ordre pour les autres nœuds en numérotant tous les messages de diffusion.

Ainsi, pour chaque groupe, un nœud sérialisateur est choisi. Tous les messages destinés au groupe lui sont adressés. Il a en charge de les diffuser au groupe de manière à ce que chacun des membres les traitent dans le même ordre. Avant d'acquitter le message à l'émetteur, le nœud sérialisateur attend qu'une majorité des membres de son groupe en ait acquitté la diffusion. En cas de défaillance du nœud sérialisateur, au moins la moitié des autres membres ont une copie à jour. De plus, dans notre mise en œuvre, les listes de membres sont ordonnées de la même manière sur chacun des membres[4] : cela permet de choisir le prochain membre non-défaillant dans la liste comme nouveau nœud sérialisateur. Le nouveau nœud sérialisateur diffuse alors un message pour récupérer les numéros de version de chaque copie afin de pouvoir re-propager la ou les mises à jour nécessaires. Durant un changement de nœud sérialisateur, le groupe est inaccessible. Quand les éventuelles mises à jour nécessaires ont été propagées, le nœud sérialisateur publie son identifiant rendant ainsi le groupe à nouveau accessible.

[3] Un mécanisme de délai de garde est utilisé afin de supporter les fautes lors de l'allocation.

[4] À l'allocation elles sont initialisées dans le même ordre, et chaque ajout/retrait s'effectue via une diffusion atomique.

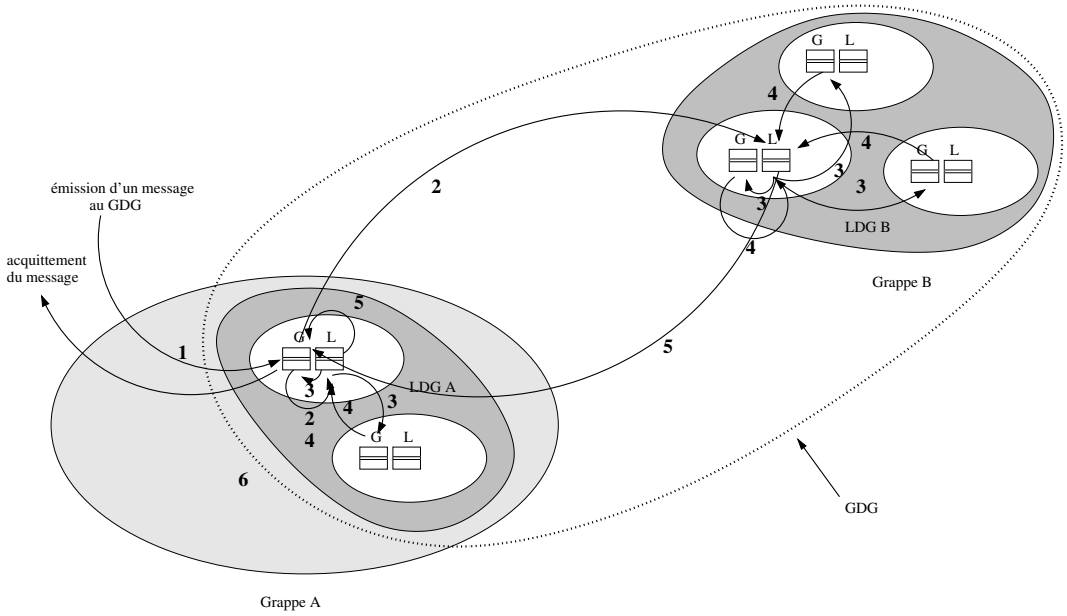

FIG. 8.7 – Diffusion des messages envoyés aux GDG.

La figure 8.7 représente la diffusion au sein des GDG. Les membres de ces groupes sont eux-mêmes des groupes : les LDG. Le mécanisme de diffusion des messages au sein des GDG est le même que celui des LDG, la diffusion au sein d'un GDG est donc hiérarchique.

1. Le nœud sérialisateur du GDG (qui est le nœud sérialisateur du LDG sérialisateur du GDG) diffuse les messages à tous ses membres, c'est-à-dire les LDG, donc les nœuds sérialisateurs des LDG (2) ;
2. les nœuds sérialisateurs des LDG diffusent les messages à leurs membres (3) et attendent des acquittements, comme décrit ci-dessus (4) ;
3. les nœuds sérialisateurs des LDG acquittent les messages dès qu'ils ont reçus suffisamment d'acquittements (5) ;
4. le GDG peut alors considérer le message comme diffusé, et l'acquitter (6).

8.3.2 Auto-organisation des groupes

En cas de faute, les groupes doivent se ré-organiser afin de pouvoir continuer à assurer les mêmes garanties de tolérance aux fautes. De nouveaux fournisseurs peuvent alors remplacer les fournisseurs défaillants.

Comme expliqué à la section 8.1.3, les groupes sont prévenus des fautes par l'intermédiaire de détecteurs de fautes. Lorsqu'une faute est détectée, le nœud sérialisateur (ou le *nouveau* nœud sérialisateur en cas de défaillance du nœud sérialisateur lui-même) diffuse de manière atomique la décision d'évincer le nœud fautif à tout le groupe. Cela permet d'assurer le changement de "vue". Une recherche est ensuite lancée afin de remplacer le fournisseur défaillant. Entre temps, le groupe continue son fonctionnement avec un membre en

moins. Quand le nouveau fournisseur est trouvé, le nœud sérialisateur gèle les communications du groupe le temps d'introduire le nouveau membre. Les requêtes arrivant sont alors mises en attente sur chacun des membres du groupe, le temps que la *réparation* se termine. Celle-ci comprend le transfert de l'état du protocole de cohérence ainsi que de la donnée sur le nouveau membre, et la diffusion de la nouvelle liste de membres à l'ensemble du groupe.

Il est ainsi possible de tolérer un très grand nombre de fautes, supérieur à la taille initiale des groupes, à condition que les occurrences de fautes ne soient pas trop fréquentes et que les groupes aient le temps de se reconfigurer.

8.4 Un protocole étendu pour une visualisation efficace

L'architecture en couches que nous avons définie permet au service JUXMEM de proposer plusieurs protocoles de cohérence ainsi que plusieurs mécanismes de tolérance aux fautes. Les clients peuvent alors sélectionner la combinaison de leur choix pour chaque donnée partagée.

Afin d'illustrer ceci, nous présentons ici un nouveau protocole de cohérence, qui constitue une extension de celui présenté à la section 8.2. Ce protocole a été conçu et mis en œuvre en collaboration avec Loïc Cudennec lors de son stage de master recherche que nous avons co-encadré. Le but de ce nouveau protocole est de permettre l'observation des données partagées en cours d'exécution de l'application les utilisant.

Les application scientifiques s'exécutant sur les grilles de calcul peuvent durer des jours, voire des semaines, et il n'est pas souhaitable d'en attendre la fin pour vérifier que tout s'est bien passé. Des données peuvent être partagées entre les codes afin de renseigner sur l'état d'avancement du calcul. L'observation d'un calcul doit s'effectuer avec des temps d'accès courts tout en causant le moins d'interactions possibles avec les autres sites pour minimiser les perturbations.

Plus spécifiquement, nous nous intéressons à l'observation des données intermédiaires dans une application à base de couplage de codes. Nous proposons donc une extension du protocole de cohérence présenté à la section 8.2 qui introduit la possibilité d'effectuer des lectures en parallèle d'une écriture en relâchant les garanties de cohérence pour ces lectures d'un genre particulier (détaillées ci-dessous). Ceci est possible si l'observateur accepte d'exploiter des données dont la version est considérée comme ancienne. Cette extension du protocole a pour conséquence une extension du modèle de cohérence utilisé. Une mise en œuvre de cette stratégie a été effectuée au sein du service JUXMEM.

8.4.1 La lecture relâchée

Le scénario considéré met en œuvre un nœud observateur dont le rôle est de lire la donnée partagée avec des temps d'accès courts sans pour autant dégrader les performances des nœuds effectuant les calculs. La première idée consiste à utiliser des copies de la donnée éventuellement anciennes, détenues par le client ou son LDG. Cette stratégie permet d'exploiter des données déjà présentes sur le client ou très proches en terme de distance réseau (au sein de la même grappe). La deuxième idée vise à diminuer les temps d'attente liés à la contention imposée par le protocole de cohérence en ne prenant plus le verrou en lecture. Une conséquence directe est d'autoriser ces lectures d'un genre particulier en même temps

que les écritures. Nous appelons ces opérations des *lectures relâchées*, et la primitive associée rlxRead. Le modèle de cohérence à l'entrée ne garantit qu'une donnée soit à jour que lors de l'acquisition de son verrou. Dans ce modèle, pour un observateur qui n'acquiert pas le verrou, le fait d'utiliser la donnée contenue dans son propre cache ou celle disponible sur son LDG ne peut garantir que la donnée soit à jour. L'approche que nous considérons propose d'autoriser ce type de lecture non-synchronisée tout en contrôlant la *fraîcheur* de la donnée. Ceci est possible en bornant l'écart entre la version retournée et la version la plus récente. Ainsi pour chaque lecture relâchée, l'application spécifie le nombre de versions de retard autorisées avant qu'une donnée ne soit considérée comme périmée.

8.4.2 Fenêtre de lecture

Pour exprimer le retard entre la version la plus récente et celle retournée par la lecture relâchée, nous introduisons deux paramètres qui prennent en compte les deux niveaux de hiérarchie du protocole de cohérence.

- La constante D, propre à chaque donnée, exprime le nombre de fois que le LDG peut transmettre successivement le verrou en écriture, sans effectuer de mise à jour du GDG. Si $D = 0$ alors le LDG doit propager les modifications après chaque relâchement du verrou en écriture. Dans ce cas, tous les LDG ont la même version de la donnée.
- Le paramètre w est spécifié par le client lors de chaque appel à la primitive de lecture relâchée rlxRead. C'est la taille de la *fenêtre de lecture*. Il exprime l'écart toléré entre la version la plus récente et celle retournée par la lecture relâchée au niveau des LDGs.

Les distances D et w sont positives ou nulles et respectent le fait que w soit supérieur ou égal à D. La différence $w - D$ correspond à la distance maximale entre la version détenue par le LDG du client et celle retournée par la lecture relâchée. A titre d'exemple, si $D = 3$ alors un LDG pourra distribuer successivement le verrou en écriture au plus 3 fois sans propager les modifications à tous les membres du GDG. Si $w = 4$, la version de la donnée lue par un client effectuant une lecture relâchée retourne soit la *dernière* version de la donnée détenue par son LDG, soit la version précédente. La notion de *dernière* version fait référence à la dernière version propagée par le LDG ayant le verrou en écriture, c'est-à-dire qu'elle peut déjà être ancienne de D versions.

8.4.3 Analyse de la sémantique des paramètres

Considérer $w = D$ implique que le client lise la même version que celle détenue par son LDG. Notons que fixer $D = 0$ et $w = 0$ ne revient pas pour autant à effectuer une lecture avec prise du verrou et ce, pour deux raisons. 1) lors de la lecture relâchée, le verrou en écriture peut de nouveau être distribué et la donnée être en cours de modification. Ceci n'est pas autorisé dans le modèle de cohérence à l'entrée. 2) Entre le moment où le LDG répond à une requête de lecture relâchée et celui où le client utilise effectivement la donnée, un nombre non contrôlé de versions a pu être produit. Cette approche convient à des applications de type visualisation mais n'offre pas de garanties strictes sur la fraîcheur des données, le contrôle de la fraîcheur est fait "au mieux" (*best effort*).

Lorsqu'un LDG (dont la version de la donnée est notée V_{LDG}) reçoit une demande de lecture relâchée (de fenêtre w) de la part d'un client C (dont la version de la donnée est V_c), la formule permettant de décider si le client peut utiliser sa version de la donnée (V_c) est :

$V_c \geq V_{\text{LDG}} - (w - D)$. L'utilisation de la donnée détenue par le client réduit le coût réseau de la lecture. Dans le cas contraire, la donnée du LDG est transférée au client. La lecture relâchée donne lieu à une extension du modèle de cohérence : le modèle de cohérence à l'entrée y est toujours respecté et garantit d'accéder à la version la plus récente de la donnée lorsque le verrou est acquis. La lecture relâchée apporte des garanties supplémentaires là où la cohérence à l'entrée n'en donne pas. Ainsi la lecture d'une valeur sans la prise du verrou n'est-elle plus complètement incontrôlée.

Les évaluations de performances que nous avons menées (voir 9.2.3) montrent que l'extension proposée ici permet d'améliorer significativement les latences des accès lors des observations d'une donnée partagée tout améliorant dans une moindre mesure celles des accès des nœuds effectuant les calculs.

8.5 Analyse

La plate-forme logicielle JUXMEM est un prototype de recherche en cours de mise en œuvre, une thèse étant en cours sur l'évolution de JUXMEM. Grâce à notre architecture logicielle, cette plate-forme permet la mise en place aisée de nouveaux protocoles de cohérence et de nouveaux mécanismes de tolérance aux fautes. Elle peut donc être utilisée comme une plate-forme expérimentale pour tester de nouveaux mécanismes et les comparer entre eux.

Dans ce chapitre nous avons donné un aperçu de la mise en œuvre de l'architecture en couches et de la manière dont il est possible de mettre en œuvre de nouveaux protocoles de cohérence et de nouveaux mécanismes de tolérance aux fautes. En utilisant l'implémentation actuelle, il est possible pour une application de choisir différents protocoles de cohérences pour des données différentes, la cohérence à l'entrée ou la cohérence à l'entrée étendue par exemple. Dans le chapitre suivant, nous montrons une évaluation comparative de ces deux protocoles dans le cas particulier de la visualisation d'une donnée partagée.

La description des mécanismes de diffusion utilisés pour la réplication des copies de référence met en évidence le coût, en terme de latence, que peuvent avoir de tels mécanismes. Les mécanismes de tolérance aux fautes risquent donc de ralentir l'application. Cependant, en présence de fautes, ces mécanismes lui permettent de continuer son exécution. Il semble important de trouver le bon compromis entre le coût des mécanismes utilisés et le risque d'apparition de fautes durant l'exécution de l'application. C'est pourquoi notre prototype permet de choisir les mécanismes utilisés ainsi que le degré de réplication. Le chapitre suivant apporte un début de réponse à la recherche de ce compromis en donnant une évaluation du coût des mécanismes de tolérance aux fautes de la plate-forme JUXMEM.

Chapitre 9

Évaluation

Sommaire

Nos approches pour la gestion conjointe de la tolérance aux fautes et de la cohérence des données au sein des grilles de calcul présentées dans ce manuscrit ont fait l'objet d'une mise en œuvre dont un aperçu est donné au chapitre 8.

Dans ce chapitre, nous donnons une évaluation de cette mise en œuvre. Dans un premier temps nous discutons de notre méthodologie d'expérimentation. En effet, l'évaluation de systèmes tolérants aux fautes pose certains problèmes, comme la difficulté d'effectuer des mesures reproductibles en présence de fautes.

Nous présentons dans la section 9.2 une évaluation de notre contribution au sein de la plate-forme JuxMem. Nous mesurons notamment le coût engendré par les mécanismes de tolérance aux fautes en absence de fautes, mais aussi le gain réalisé grâce à notre approche hiérarchique pour la mise en place de protocoles de cohérence tolérants aux fautes. Une évaluation comparative de deux protocoles de cohérence illustre l'intérêt d'une plate-forme multi-protocole. Enfin, nous mesurons le surcoût engendré en cas d'occurrences de fautes.

9.1 Méthodologie d'expérimentation

Il existe de multiples méthodes permettant d'évaluer des systèmes : les preuves formelles, les simulations et enfin les expérimentations. Ces méthodes ne présentent pas toutes les mêmes avantages et inconvénients.

Preuves formelles et simulations. Ces deux méthodes présentent l'inconvénient d'évaluer un *modèle* du système et *non le système lui-même*[1]. Les modèles de systèmes sont une simplification des systèmes réels et leur évaluation ne donne qu'une idée du comportement du système réel. En revanche, les preuves formelles permettent souvent de prendre en compte *tous* les cas, ce qui est rarement le cas pour les simulations ou les expérimentations sur les systèmes réels. Les simulations quant à elles présentent l'avantage non négligeable de pouvoir être réalisées assez rapidement.

9.1.1 Expérimentations sur architectures réelles

Les expérimentations de systèmes sur des architectures réelles permettent d'évaluer le comportement du système dans des conditions réelles. De plus, elles permettent aux développeurs de mettre au point les systèmes et de les optimiser. Ce type d'évaluation est néanmoins plus difficile et plus long à mettre en œuvre. Dans le cas des expérimentations sur des plates-formes distribuées telles que les grilles de calcul, il est généralement nécessaire d'utiliser des outils de réservation de ressources (nœuds), de déployer le système sur les nœuds obtenus, de lancer le système et enfin de récupérer les résultats *répartis* sur l'ensemble des nœuds utilisés. Dans ce contexte, expérimenter des mécanismes de tolérance aux fautes ajoute à la difficulté : il devient alors nécessaire de pouvoir *contrôler* les fautes afin d'évaluer les mécanismes mis en place pour y faire face. Nous avons expérimenté notre service de partage de données pour la grille JUXMEM sur la plate-forme Grid'5000 [113, 26]. Pour ce faire, nous avons utilisé des prototypes de recherche pour les réservations de nœuds (OAR [121] et OARGRID [122]) ainsi que pour le déploiement (ADAGE [74]).

OAR et OARGRID. OAR est un outil de gestion de ressources fondé sur MySQL [120]. Il permet de réserver des nœuds et de lancer des exécutions au sein d'une grappe. Cet outil est présent sur chacune des grappes de Grid'5000. OARGRID utilise OAR sur plusieurs grappes afin d'offrir une gestion de ressources au niveau de la grille.

ADAGE est un outil de déploiement générique d'applications distribuées et/ou parallèles s'exécutant sur les grilles de calcul. ADAGE dispose d'un greffon (en anglais *plugin*) pour le déploiement d'applications basées sur JXTA [64].

Nous avons mis en œuvre un outil d'injection de fautes permettant d'évaluer la réaction aux fautes de notre mise en œuvre. Cet outil est présenté à la section suivante.

[1] Même si certains simulateurs permettent d'exécuter le code du système à expérimenter, l'architecture matérielle utilisée est modélisée et simulée.

9.1.2 Injection de fautes de type panne franche

L'évaluation des mécanismes de tolérance aux fautes requiert des expérimentations en *présence de fautes*. En effet, des exécutions sans fautes permettent uniquement de mesurer le surcoût éventuel que ces mécanismes engendrent. Notre but est donc de pouvoir engendrer et contrôler les fautes survenant dans le système.

Nous considérons que la plate-forme de test est stable (sans fautes). En cas de faute réelle (non contrôlée/injectée), le test est considéré comme avorté et doit être relancé. Nous avons mis au point un outil permettant d'injecter des défaillances franches de nœuds. Les propriétés offertes par cet outil sont la reproductibilité, la précision et le passage à l'échelle.

Reproductibilité. Comparer des mécanismes de tolérance aux fautes dans des conditions similaires nécessite de pouvoir également reproduire les conditions de fautes. Il semble donc judicieux qu'un injecteur de fautes offre la propriété de reproductibilité. Cela peut également permettre d'effectuer de multiples fois le même test afin d'obtenir une valeur moyenne, un écart-type, etc.

Notre outil calcule un calendrier de fautes *avant* le déploiement du système. Ce calendrier peut ainsi être utilisé de multiples fois par la suite.

Précision. Au sein d'un système distribué, tous les nœuds ne jouent pas nécessairement le même rôle. Nous avons notamment décrit notre mécanisme de réplication basé sur un nœud sérialisateur au chapitre 8. La défaillance d'un nœud sérialisateur n'est pas gérée de la même manière que la défaillance d'un nœud non-sérialisateur : la première entraîne notamment le choix d'un nouveau nœud sérialisateur pour le groupe de réplication. Un injecteur de fautes doit pouvoir permettre de déterminer finement quels nœuds vont être défaillants et quand.

Notre outil est intégré dans l'environnement de test JDF (*JXTA Distributed Framework* [114]) qui utilise la notion de *profil* (représentant des nœuds ou groupes de nœuds) afin de décrire efficacement l'ensemble des nœuds intervenant lors du test. Nous avons étendu le langage de description de cet environnement afin de permettre la description de dépendances, entre profils. Cela offre la possibilité d'injecter des fautes corrélées. Par exemple tous les nœuds d'un même profil peuvent être défaillant simultanément. Le calendrier calculé peut ensuite être raffiné nœud par nœud.

Passage à l'échelle. Définir une date de défaillance pour chaque nœud, un par un, peut s'avérer long lorsque le nombre de nœuds est grand. Afin de pouvoir générer un calendrier de fautes pour de nombreux nœuds, notre injecteur de fautes peut s'appuyer sur des distributions statistiques, typiquement, une distribution exponentielle paramétrée par un temps interfaute moyen (MTBF pour *Mean Time Between Failures*). Notons que l'utilisation d'une telle distribution ne nuit en rien à la propriété de reproductibilité : le calendrier est calculé en utilisant la distribution puis peut être utilisé à volonté.

Plus de détails sur notre outil d'injection de fautes peuvent être trouvés dans [MB06] et [ABJM04].

9.2 Expérimentations avec JUXMEM

L'ensemble des expérimentations décrites ci-dessous a été réalisé sur la plate-forme Grid'5000 [113, 26], notamment sur les grappes de Nancy ($G1$), Orsay ($G2$) et Rennes ($G3$).

Dans l'ensemble de ces grappes, les machines utilisées sont équipées de bi-processeurs Opteron d'AMD cadencés à 2,0 GHz, munis de 2 Go de mémoire vive et exécutant la version 2.6 du noyau Linux. Le réseau utilisé au sein de chacune de ces grappe est Gigabit Ethernet. Les grappes sont reliées entre elles par le réseau Renater [125]. Les latences entre ces différentes grappes sont représentées par le tableau 9.1.

Rennes - Nancy	Rennes - Orsay	Nancy - Orsay
6 μs	4.5 μs	3 μs

TAB. 9.1 – Latence entre les grappes utilisées.

Toutes les expérimentations ont été réalisées avec la version C de JUXMEM à l'exception de celles de la section 9.2.3 qui ont été réalisées avec la version Java pour des raisons de disponibilité de code. Pour ces expérimentations, chaque nœud héberge un seul pair JUXMEM. Bien que JUXMEM soit un service prévu pour être utilisé par des applications partageant de nombreuses données sur de nombreux nœuds, pour toutes les évaluations présentées ici, nous observons le comportement de JUXMEM au niveau d'*une* donnée partagée. Le nombre de nœuds intervenant dans les expérimentations est donc limité.

9.2.1 Coût dû à la réplication

Le but de cette première série d'expérimentations est d'évaluer le surcoût dû à la tolérance aux fautes tel qu'observé par les applications (les clients JUXMEM). Nous mesurons donc les coûts des accès à une donnée partagée pour différents degrés de réplication et différentes tailles de données. Le type de réplication utilisé est *semi-actif*, il correspond à celui fondé sur un nœud sérialisateur décrit au chapitre précédent.

Description. Nous utilisons ici 3 grappes dans lesquelles nous créons 3 groupes *cluster* JUXMEM. Chacun des groupes *cluster* contient 1 pair gestionnaire et 7 pairs fournisseurs. Le groupe *cluster* correspondant à la grappe *G*3 localisée à Rennes héberge de plus 1 pair client. Afin d'éviter le "bruit" qui serait introduit dans nos mesures par des attentes sur des verrous, ce pair est le seul a accéder à la donnée, séquentiellement, alternant lectures et écritures. Afin que les lectures engendrent les transferts de données, ce client projette (*map*) 2 fois la donnée dans sa mémoire locale : une fois pour les lectures et une fois pour les écritures. Tout se passe donc comme s'il y avait 2 clients parfaitement synchronisés effectuant des lectures et écritures à tour de rôle.

Une *lecture* correspond à la prise du verrou en lecture (acquire_read), la récupération de la valeur de la donnée et à la libération du verrou (release). Une *écriture* correspond à la prise du verrou en écriture (acquire), à l'écriture de la donnée (memset) et à la libération du verrou avec propagation de la donnée (release).

Nous mesurons les performances des lectures et écritures avec des données de taille comprise entre 1 ko et 64 Mo (1 Mo dans la majorité des cas) en faisant varier la taille du GDG (*Global Data Group*) de 1 à 3 LDG (*Local Data Group*), et la taille des LDG de 1 à 7 fournisseurs (figures 9.1, 9.2, 9.3, 9.4, 9.5 et 9.6). Chaque courbe représente les latences pour un degré de réplication, noté sous la forme "nombre de grappes" $\times$ "nombre de fournisseurs par grappe".

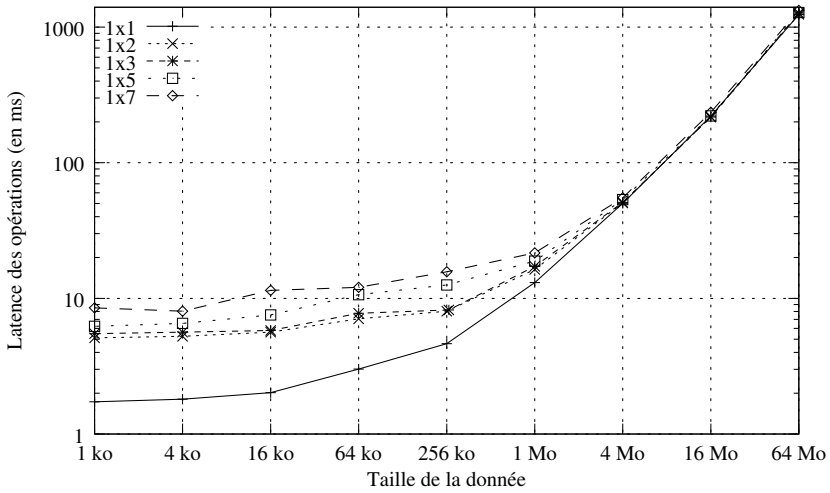

FIG. 9.1 – Latences des opérations de lecture en fonction de la taille de la donnée et du nombre de fournisseurs dans une grappe.

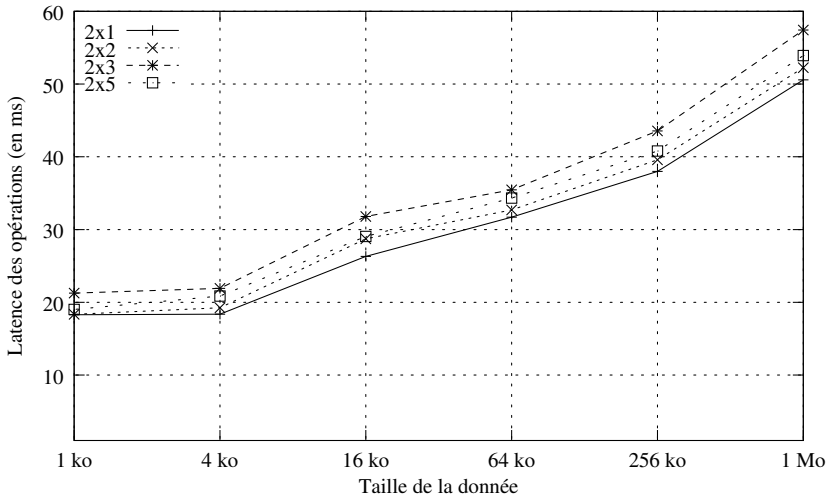

FIG. 9.2 – Latences des opérations de lecture en fonction de la taille de la donnée et du nombre de fournisseurs répartis dans 2 grappes.

Lectures. Les figures 9.1, 9.2 et 9.3 présentent les latences des lectures d'une donnée répliquée dans respectivement 1, 2 et 3 grappes. Ces figures montrent que le surcoût de la réplication au sein des grappes (LDG) sur les opérations de lecture de données de grande taille est restreint : il est inférieur à 3% dès que la taille de la donnée dépasse 1 Mo. Pour les données de petites taille, ce surcoût est toutefois important : 4 ms pour une donnée de 1 ko. Ces résultats s'expliquent par le fait que les opérations de lecture ne font intervenir les mécanismes de réplication que pour les opérations de synchronisation acquire_read et release. La donnée est transférée à partir d'un seul nœud fournisseur. Aussi, plus la donnée est grande, plus le surcoût introduit par la réplication lors des opérations de synchronisation est négligeable par rapport au temps de transfert de la donnée du fournisseur au client. En revanche pour les données de petite taille, le coût engendré par les échanges entre les

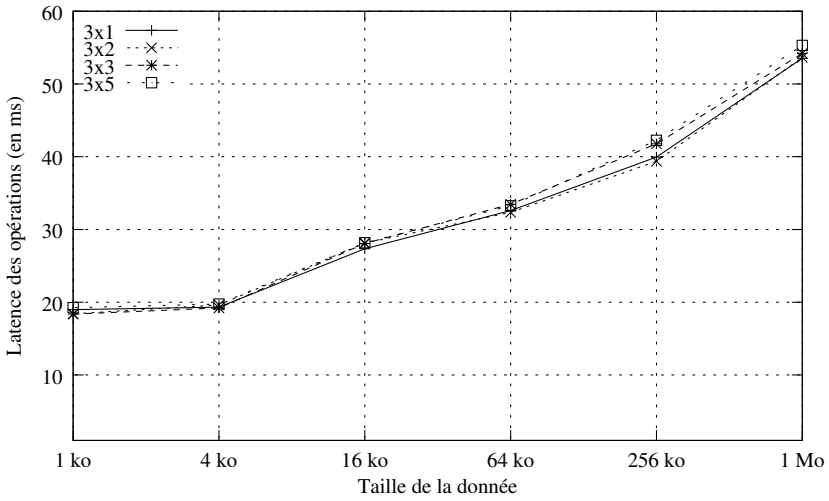

FIG. 9.3 – Latences des opérations de lecture en fonction de la taille de la donnée et du nombre de fournisseurs répartis dans 3 grappes.

fournisseurs lors des opérations de synchronisation est non négligeable. Les opérations de synchronisation acquire_read et release engendrent des communications de groupe comme décrit à la section 8.3.

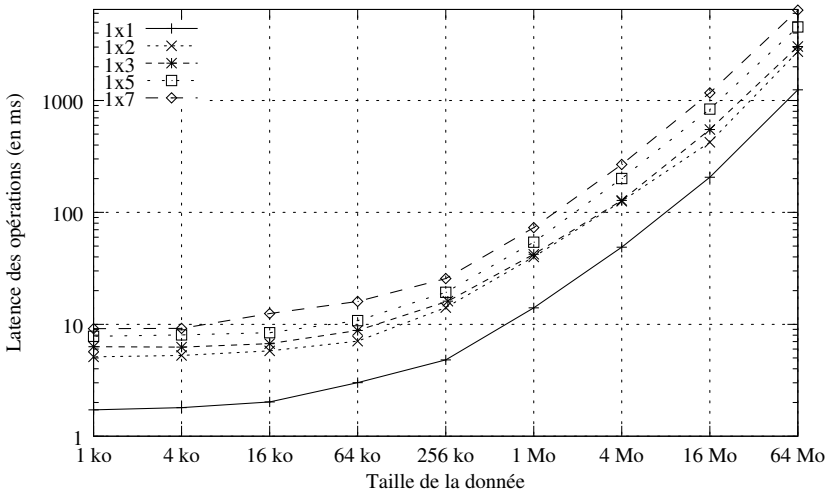

FIG. 9.4 – Latences des opérations d'écriture en fonction de la taille de la donnée et du nombre de fournisseurs dans une grappe.

Écritures. Les figures 9.4, 9.5 et 9.6 présentent les latences des écritures d'une donnée répliquée dans respectivement 1, 2 et 3 grappes. Pour les opérations d'écriture, le coût de la réplication au sein d'une grappe est élevé : le simple ajout d'une copie fait plus que doubler les temps d'écriture de la donnée (figure 9.4, courbes 1×1 et 1×2). Ce surcoût est essentiellement dû au temps de transfert de la donnée sur le deuxième fournisseur et à l'attente de son acquittement : la donnée est d'abord transférée du client vers le fournisseur jouant le rôle de

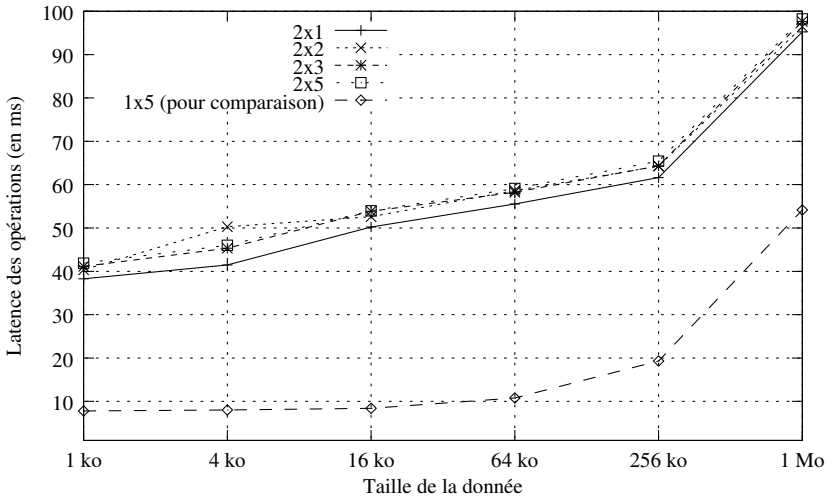

FIG. 9.5 – Latences des opérations d'écriture en fonction de la taille de la donnée et du nombre de fournisseurs répartis dans 2 grappes.

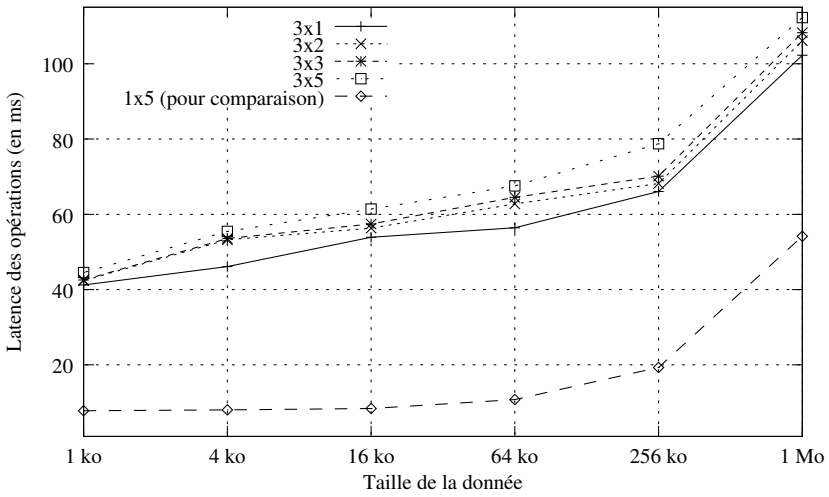

FIG. 9.6 – Latences des opérations d'écriture en fonction de la taille de la donnée et du nombre de fournisseurs répartis dans 3 grappes.

sérialisateur, puis de ce premier fournisseur vers le deuxième. En revanche, l'ajout de copies supplémentaires est bien moins coûteux. Le nœud sérialisateur n'attend pas d'acquittement de *tous* les membres, mais seulement *d'une majorité* d'entre eux. Ainsi, dans le cas 1×3, un acquittement peut-être retourné au client avant que la donnée n'est été transférée complètement sur le troisième fournisseur. Lors de nos expérimentations, nous avons observé une différence de l'ordre de 5% pour des données de 1 Mo entre l'utilisation de n et de $n + 1$ fournisseurs, pour $n > 1$.

Les figures 9.5 et 9.6 montrent que le surcoût de la réplication intragrappe est négligeable par rapport au surcoût de la réplication intergrappe. En effet, les mécanismes de réplication intergrappe utilisent les liens réseau reliant les grappes entre elles. Ces liens ont une latence bien plus élevée que ceux présents au sein des grappes. Cependant, la série d'expérimenta-

tion suivante montre l'intérêt de la présence de copies de la donnée dans chaque grappe où des clients sont susceptibles d'y accéder.

9.2.2 Bénéfice de l'approche hiérarchique

Notre approche hiérarchique présentée au chapitre 7 permet de limiter l'utilisation des liens réseau intergrappe au profit des liens réseau intragrappe de plus faible latence. Elle permet à chaque client d'avoir accès à une copie de référence locale située dans la même grappe. Le but de cette série d'expérimentations est d'évaluer le bénéfice d'une telle approche. Pour réaliser ces tests, nous faisons correspondre un groupe *cluster* JUXMEM à l'ensemble des trois grappes afin de masquer la hiérarchie de la grille au niveau de notre architecture en couches.

Intérêt des copies de référence locales. Cette expérimentation à pour but de mettre en évidence l'intérêt de la présence d'une copie de référence locale dans la grappe d'un client accédant à la donnée hébergée par cette copie. Nous supprimons donc les fournisseurs situés dans la même grappe que le client ($G3$) et mesurons le coût des lectures et écritures du client lorsque le GDG ne contient qu'un LDG lui même composé d'un seul fournisseur. Le client accède donc à une copie de référence non répliquée et non hiérarchique située dans une grappe distante ($G1$ lors de ces tests). Les mesures sont présentées par la figure 9.7.

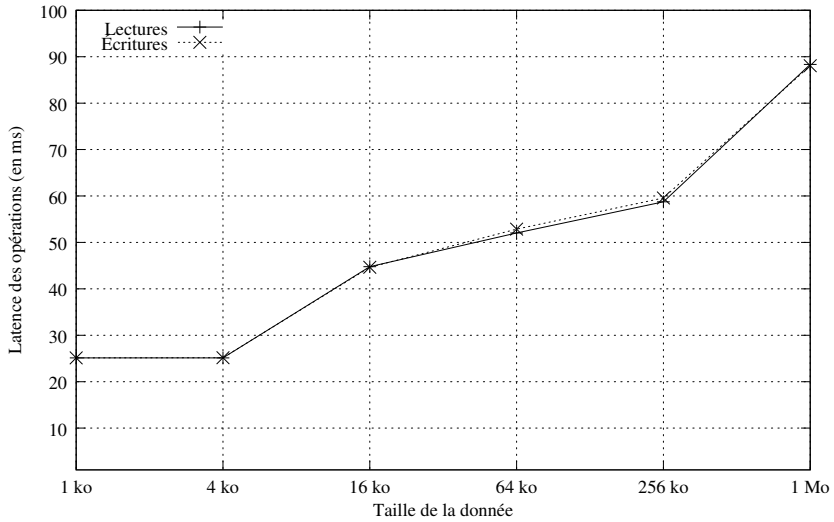

FIG. 9.7 – Latences des opérations faisant intervenir un fournisseur distant en fonction de la taille de la donnée.

Si l'on compare ces résultats avec ceux obtenus lors de la série de test précédente lorsqu'un fournisseur unique (cas 1×1) présent dans la grappe $G3$ hébergeait la copie de référence, on s'aperçoit que la latence des accès aussi bien en lecture qu'en écriture est augmentée. Le tableau 9.2.2 reprend les temps de lecture et d'écriture pour ces deux cas. Les latences des écritures et celles des lectures étant semblables dans les deux cas, nous présentons ici une moyenne.

Taille de la donnée en ko	1	4	16	64	256	1024
Temps moyen d'accès en ms (cas local)	1.72	1.80	2.02	3.01	4.72	13.54
Temps moyen d'accès en ms (cas distant)	25.12	25.13	44.76	52.04	58.74	88.35

TAB. 9.2 – Comparaison des temps d'accès lorsque la copie de référence est locale (1×1) ou distante.

Cette différence est due à l'augmentation de la latence réseau entre le client et le fournisseur hébergeant la copie de référence (6 ms entre Rennes et Nancy hébergeant respectivement $G3$ et $G1$). Les opérations de lecture ou d'écriture requièrent 2 échanges entre le client et la copie de référence : 1) une requête et son acquittement pour la prise de verrou ; et 2) une requête et son acquittement pour la libération du verrou (la donnée est encapsulée dans les requêtes et/ou acquittements lorsque cela est nécessaire). Ces 4 échanges sur un lien réseau de latence 6 ms impliquent une latence des opérations au moins supérieure à $6\,ms * 4 = 24\,ms$.

Si l'on ajoute une copie de référence locale dans la grappe du client en conservant la copie présente dans la grappe distante et en utilisant notre approche hiérarchique, on obtient les résultats du cas 2×1 (figures 9.2 et 9.5), rappelés dans le tableau 9.2.2.

Taille de la donnée en ko	1	4	16	64	256	1024
Lectures en ms (cas 2×1)	18.29	18.36	26.31	31.69	38.00	50.58
Écritures en ms (cas 2×1)	38.29	41.47	50.21	55.55	61.63	95.25
Temps moyen d'accès en ms (cas distant)	25.12	25.13	44.76	52.04	58.74	88.35

TAB. 9.3 – Comparaisons des temps d'accès lorsqu'il y a un seul fournisseur distant ou un fournisseur distant et un local.

En plaçant une copie de référence locale proche du client, en utilisant notre approche hiérarchique, les performances des lectures se retrouvent améliorées au détriment de celles des écritures. La copie de référence locale permet en effet des lectures de la donnée sur un fournisseur proche en terme de latence, cependant la mise en service des mécanismes de réplication entre les deux grappes, par la présence de deux copies de la donnée, ralentie légèrement les écritures. En effet, chaque écriture entraîne ici une mise à jour de la copie distante. Notons que la présence de plusieurs copies de référence permet par ailleurs d'offrir de meilleures garanties de tolérance aux fautes.

Intérêt des groupes hiérarchiques. Le but de cette série d'expériences est de mettre en évidence l'intérêt de l'approche hiérarchique des groupes. Nous mesurons le coût des accès à une donnée partagée par plusieurs clients, répartis dans 2 sites différents : 2 s'exécutent dans la grappe $G1$ à Nancy et 2 s'exécutent dans la grappe $G3$ localisée à Rennes. Chacun de ces clients effectue des itérations au sein desquelles il lit la donnée partagée, opère une phase de calcul puis modifie complètement la donnée partagée. Lors de ces mesures la donnée est répliquée sur 6 fournisseurs, 3 dans $G1$ et 3 dans $G3$. Nous observons alors deux cas : 1) avec hiérarchie, c'est-à-dire que $G1$ et $G3$ hébergent chacune un LDG formé de 3 fournisseurs et composant un GDG ; et 2) sans hiérarchie, c'est-à-dire que l'ensemble des 6 fournisseurs

forme un unique LDG. Pour ce deuxième cas nous "cachons" la topologie physique en configurant nos tests de manière à ne créer qu'un seul groupe *cluster* composé de l'ensemble des nœuds des 2 grappes. La figure 9.8 illustre ces deux configurations.

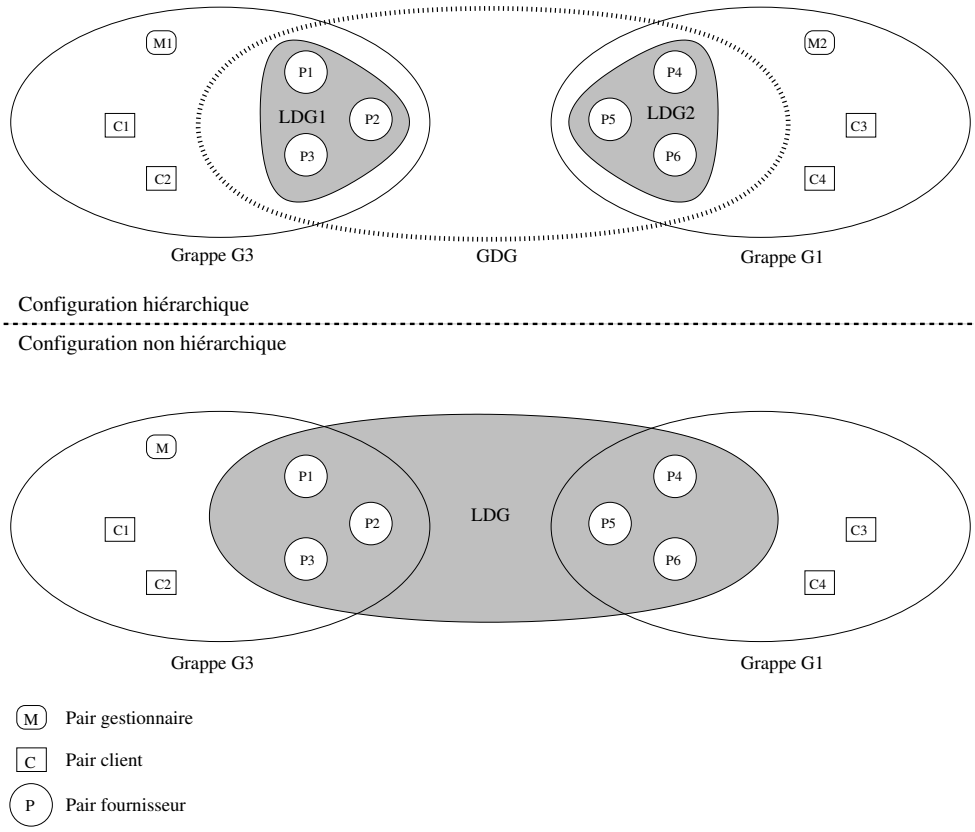

FIG. 9.8 – Quatre clients répartis dans deux grappes accèdent à une données répliquée sur six fournisseurs.

La figure 9.9 présente les latences des lectures et des écritures dans les 2 cas, en fonction de la taille de la donnée. Les latences données sont les moyennes des temps d'accès de chacun des clients. Ces résultats montrent que dans le cas où l'on utilise pas l'approche hiérarchique, les latences des opérations d'écriture et de lecture sont nettement détériorées par rapport au cas où l'on utilise cette approche. Cette détérioration des performances provient : 1) de *la réplication "à plat"* entre les deux grappes ; et 2) de la distance entre certains clients et le fournisseur jouant le rôle de sérialisateur. La réplication "à plat" est moins performante car le nœud sérialisateur du LDG attend qu'une majorité de ses membres ait acquitté la diffusion. Dans le cas présent il attend 3 acquittements, donc au moins un en provenance d'un fournisseur présent dans la deuxième grappe. De plus la moitié des clients sont situés dans une grappe différente de celle contenant le fournisseur sérialisateur. Les accès à la donnée par ces clients vont donc engendrer des communications à plus forte latence et détériorer les performances, comme nous l'avons remarqué lors du test précédent. Au vu de ces résultats, l'approche hiérarchique pour la conception de protocoles de cohérence tolérants aux fautes semble bien adaptée aux architectures de type fédération de grappes.

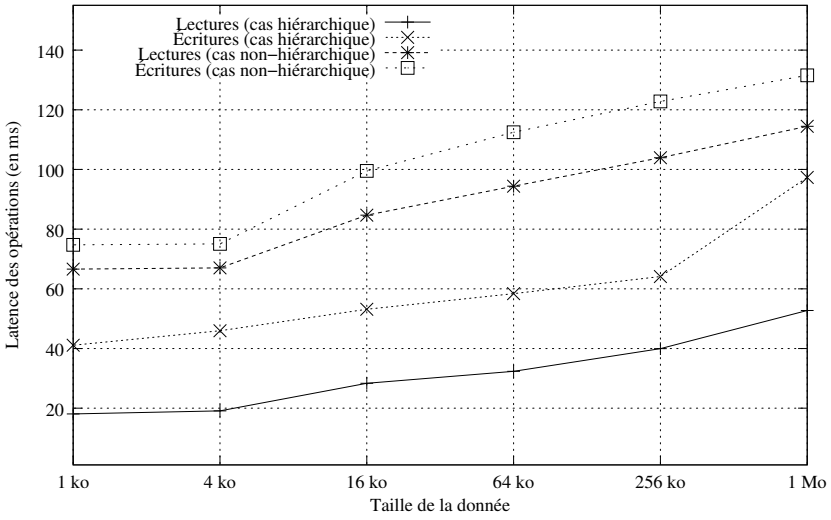

FIG. 9.9 – Comparaison des latences des opérations avec et sans notre approche hiérarchique.

9.2.3 Évaluations multi-protocole

JUXMEM est une plate-forme multi-protocole, c'est-à-dire qu'il est possible à l'exécution de choisir entre différents protocoles de cohérence ou mécanismes de tolérance aux fautes, et ceci pour chaque donnée partagée. Les expérimentations présentées ici ont pour but d'illustrer le bénéfice qu'il est possible de tirer de cette possibilité. Ces expérimentations ont été effectuées avec la version Java de JUXMEM pour laquelle ces deux protocoles sont disponibles. Pour ces expérimentations, nous avons utilisé les grappes du projet Grid'5000 localisées à Orsay ($G2$), Rennes ($G3$) et Toulouse ($G4$).

Nous reprenons le protocole de cohérence à l'entrée, présenté dans la section 8.2 et le protocole étendu pour une visualisation efficace présenté dans la section 8.4. Le but de ces expérimentations est d'évaluer les bénéfices apportés par l'extension du modèle de cohérence à l'entrée. Une évaluation détaillée de cette extension est présentée dans [ACM06b], nous présentons ici celles mettant en évidence les bénéfices de cette extension pour les opérations d'observations de données partagées.

Description. Nous considérons une application synthétique de type couplage de codes s'exécutant sur deux grappes différentes : une à Rennes et une à Toulouse. Au sein de la grappe $G3$ (Rennes), un client met à jour la valeur d'une donnée partagée de manière itérative. La grappe $G4$ (Toulouse) héberge un client effectuant des opérations de lecture de cette même donnée. Au sein de la troisième grappe localisée à Orsay, un processus *observateur* est en charge d'observer l'évolution de la valeur de la donnée partagée. Comme nous l'avons souligné à la section 8.4, l'observateur ne requiert pas nécessairement les mêmes garanties de cohérence que les processus de l'application-même : il peut se contenter de versions légèrement obsolètes de la donnée. Pour réaliser ce test nous avons utilisé 3 copies de la donnée : une copie dans chacune des grappes.

La figure 9.10 présente les latences des opérations d'observation effectuées par l'observateur localisé à Orsay. Pour 3 tailles de données, les latences des observations sont mesurées

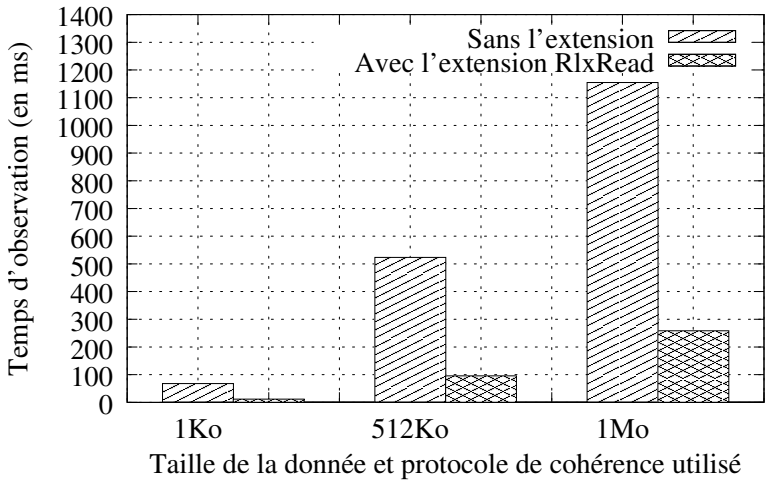

FIG. 9.10 – Comparaison de deux protocoles de cohérence lors d'opérations d'observation.

avec le protocole implémentant la cohérence à l'entrée et avec le protocole implémentant *l'extension* du modèle de cohérence à l'entrée présentée à la section 8.4. Les paramètres w (nombre maximal de versions de retard acceptées par l'observateur) et D (nombre maximal de redistribution du verrou dans une grappe sans mise à jour de la donnée dans les autres grappes) sont tous deux égal à 0. Comme détaillé à la section 8.4 cela signifie que que le client observe la version de la donnée détenue par son LDG au moment où il effectue la requête, et que le LDG est mis à jour chaque fois qu'un verrou en écriture est relâché dans le système. Les latences des opérations rlxRead du protocole étendu sont 4 à 6 fois plus rapides que les observations du protocole sans extension, utilisant les opérations acquire_read et release. Ce gain en performance s'explique par le fait que les opérations rlxRead ne sont jamais bloquées par les prises de verrou des autres clients, la donnée peut être observée alors que des écritures sont en cours. De plus, lors de ces expériences, nous avons mesuré un gain pouvant atteindre 30% sur les opérations effectuées par les clients de l'application. L'opération rlxRead ne nécessite en effet pas de prise de verrou et ne bloque donc jamais les opérations des clients des autres grappes. En revanche le verrou est indisponible pour les autres clients entre les opérations acquire_read et release.

Des évaluation plus détaillées [ACM06b] montrent qu'il est encore possible d'améliorer les performances des observations en relâchant les paramètres D et w, c'est-à-dire en acceptant des valeurs plus anciennes de la donnée.

9.2.4 Impact des fautes sur les performances

Au chapitre 8, nous avons décrit le comportement des SOG (groupes auto-organisants) face aux fautes. Il faut noter qu'au moment de l'occurrence d'une faute, les performances ne sont pas dégradées : le groupe continue à être disponible avec les membres restants. Quand une faute est détectée, un nouveau fournisseur est recherché. C'est seulement lorsque le nouveau fournisseur a été trouvé que le groupe "gèle" ses communications et que les performances des accès se trouvent affectées.

L'objectif des ces expérimentations est d'évaluer : 1) le temps pendant lequel un groupe

est en sous-nombre après l'occurrence d'une faute ; et 2) le temps pendant lequel un groupe gèle ses communications.

Description. Nous mesurons trois temps : 1) le temps de détection d'une faute ; 2) le temps de recherche d'un nouveau fournisseur ; et 3) le temps de gel du groupe, c'est-à-dire le temps de réparation qui consiste à mettre à jour l'état du nouveau fournisseur (comprenant le transfert de la donnée) et les listes de membres de l'ensemble du groupe. Ce troisième temps correspond à la durée durant laquelle le groupe ne répondra pas aux messages qui lui sont envoyés (il sont stockés sur le nœud sérialisateur en attente de la fin de la réparation). La somme de ces trois temps représente la *durée de vulnérabilité* du groupe : temps pendant lequel le groupe possède un nombre de membres restreint.

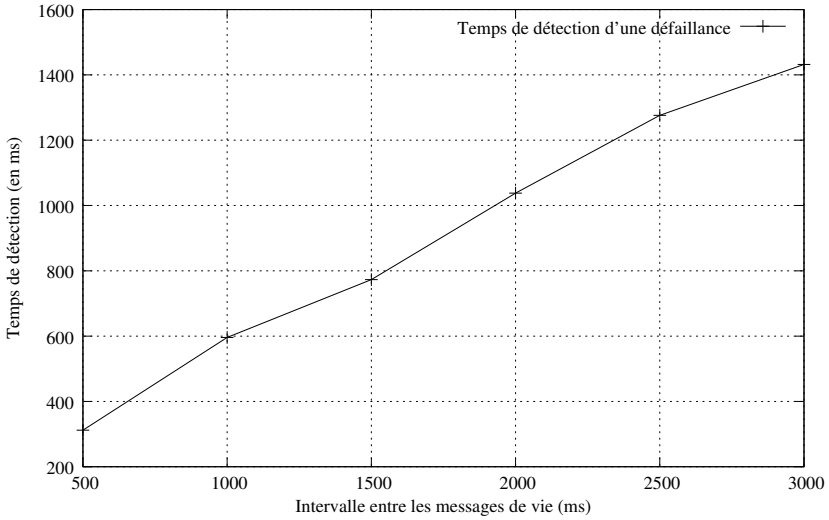

FIG. 9.11 – Temps de détection en fonction de l'intervalle entre les messages de vie.

Détection de fautes. Nous avons évalué le détecteur de fautes hiérarchique [16] présenté à la section 3.2 grâce à notre outil d'injection de fautes. Le temps de détection est évalué sur 64 nœuds uniformément répartis dans 4 grappes du projet Grid'5000 : Rennes, Lyon, Grenoble et Sophia Antipolis. Pour cette expérimentation nous avons utilisé la version Java de JUXMEM. Le temps de détection d'une faute dépend du réglage du détecteur. Le principal paramètre est l'intervalle séparant l'émission de deux messages de vies. La figure 9.11 présente les temps de détection au sein d'une grappe en fonction de cet intervalle. Il sont en moyenne égaux à la moitié du temps écoulé entre deux message de vie. Nous avons également utilisé la possibilité d'injecter des fautes corrélées afin d'évaluer la détection de fautes entre les grappes. Nous obtenons des résultats similaires. La figure 9.11 peut être utilisée pour choisir le bon compromis entre vitesse de détection et nombre de messages utilisés.

Une description approfondie de ces expérimentations ainsi que des résultats détaillés sont présenté dans [MB06].

Recherche d'un nouveau fournisseur. Les expérimentions ont été effectuées au sein de la grappe de Rennes, en utilisant la version C de JUXMEM. La configuration utilisée met en

œuvre un gestionnaire, 4 fournisseurs et un client. Le client alloue une donnée répliquée sur trois fournisseurs, après un laps de temps, un des fournisseurs est considéré comme défaillant, il est remplacé par un appel à la fonction `int repair(Jxta_id *faulty_peer_id)`, fonction normalement appelée par le détecteur de fautes.

Pour la recherche du nouveau fournisseur, nous nous appuyons sur le noyau de JUXMEM. Les mécanismes mis en œuvre lors de cette recherche sont détaillés dans [64]. Afin de trouver un fournisseur différent de ceux déjà présent dans le groupe, nous recherchons 4 fournisseurs. Lors de nos expérimentations nous avons mesuré un temps de recherche de 4.6 ms. Ce temps est indépendant de la taille de la donnée.

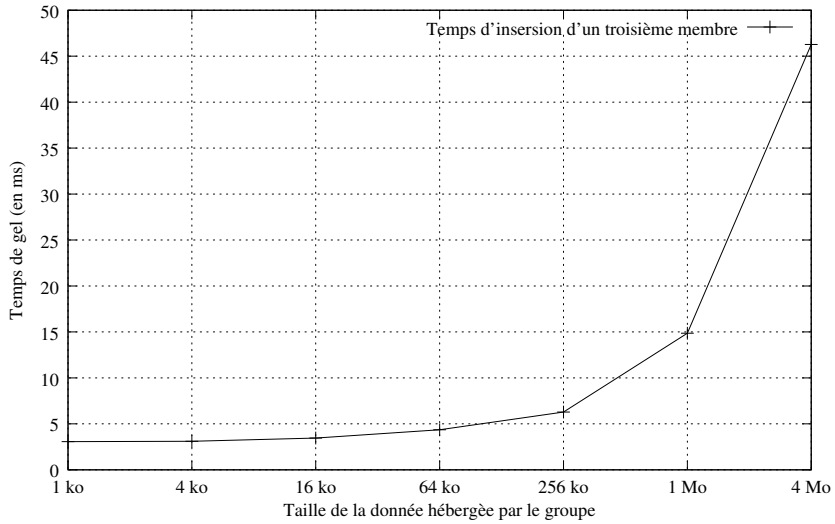

FIG. 9.12 – Temps de gel d'un groupe lors de l'insertion d'un nouveau membre.

Insertion du nouveau fournisseur dans le groupe. Pour mesurer le temps d'insertion du nouveau fournisseur dans le groupe, nous reprenons la configuration du test précédent. La figure 9.12 présente les temps d'insertion d'un nouveau fournisseur dans un groupe de 2 fournisseurs en fonction de la taille de la donnée. Ce temps comprend l'envoi de la donnée et de l'état du protocole de cohérence au nouveau membre, ainsi que l'envoi de la nouvelle liste de membres à l'ensemble des membres du groupe. Le protocole de cohérence reste gelé jusqu'à réception des acquittements des membres du nouveau groupe. On remarque que ce temps est directement lié au temps de transfert de la donnée, donc à sa taille.

9.3 Discussion

Les évaluations présentées ici montrent que les mécanismes de tolérance aux fautes ont un coût important lorsque l'on considère une donnée accédée au sein d'*une seule grappe*. Cependant, elles ont également montré que lorsque des clients situés dans *des grappes différentes* accèdent à une même donnée partagée, la présence d'une copie de référence locale dans chaque grappe pouvait améliorer significativement les performances des accès des différents clients. De plus dans ces cas-là, la réplication les copies de référence locales au sein

de leur grappe présente un coût relativement bas.

Les expérimentations comparant une approche non-hiérarchique avec notre approche hiérarchique mettent en évidence l'intérêt de l'approche hiérarchique pour les performances des accès aux données partagées sur les grilles de type fédération de grappes.

Enfin, JUXMEM est une plate-forme multi-protocole permettant notamment de comparer différents protocoles de cohérence. Ainsi, nous avons pu évaluer les bénéfices apportés par l'extension du protocole de cohérence présentée dans la section 8.4.

Quatrième partie

Conclusion et perspectives

Chapitre 10

Conclusion et perspectives

Conclusion

Contexte d'étude

Les travaux présentés dans ce manuscrit s'inscrivent dans le contexte des grilles de calcul. Les grilles de calcul s'imposent comme une solution très intéressante face à la demande croissante de puissance de stockage et de calcul des applications scientifiques. Elles permettent en effet d'agréger des ressources informatiques afin d'additionner leur espace de stockage et leur puissance de calcul. Pour obtenir une grille plus puissante, il "suffit" donc d'ajouter des ressources, c'est-à-dire des nœuds, des grappes de nœuds ou des calculateurs parallèles. La vision ultime est que l'utilisation de la puissance offerte par toutes ces ressources agrégées sera à terme aussi simple que l'utilisation de la puissance électrique fournie en branchant une prise.

Toutefois, les grilles de calcul ne sont pas encore arrivées à ce niveau de maturité. À ce jour, l'utilisation d'une grille reste complexe et réservée à des spécialistes. Les principales difficultés proviennent de la nature même des grilles de calcul : ce sont des architectures *hétérogènes*, *à grande échelle* et *dynamiques*.

Hétérogénéité. Les grilles permettent d'agréger des ressources de différentes organisations, mais ces ressources ne sont généralement pas homogènes. Les nœuds peuvent avoir des architectures matérielles différentes (par exemple 32 bits/64 bits), les systèmes d'exploitation des nœuds ainsi que les différentes bibliothèques logicielles ou outils disponibles peuvent varier. Enfin, les équipements réseaux sont eux également très hétérogènes (Ethernet, Gigabit-Ethernet, Myrinet, etc.). Cette hétérogénéité rend complexe l'utilisation des grilles de calcul.

Grande échelle. Le nombre de ressources composant une grille de calcul peut s'avérer très grand, Les projets actuels de grilles de calculs comme Grid'5000 [113, 26], TeraGrid [126] ou EUROGRID [109] sont composés de milliers voire de dizaines de milliers de nœuds. De surcroît, ces ressources sont distribuées géographiquement à l'échelle de pays, voire de continents.

Dynamicité. Cette difficulté découle en partie de la grande échelle. Plus le nombre de ressources utilisées est grand, plus la possibilité d'occurrence de fautes est importante. À l'échelle des grilles de calcul, les fautes sont donc fréquentes. De plus, la complexité des réseaux utilisés peut aboutir à des fautes mettant en péril la connectivité, une grille pouvant alors se retrouver partitionnée. Une partie de la dynamicité est engendrée par des opérations de maintenance sur des nœuds ou sur l'ensemble des ressources d'une organisation.

Ces propriétés rendent difficile la mise en œuvre et le déploiement d'applications sur les grilles de calcul. De nombreux travaux de recherche ont proposé des solutions permettant de faire face à une partie de ces difficultés et de d'autres sont encore en cours. Des projets comme Globus [112] proposent un ensemble d'outils facilitant l'exploitation des grilles, notamment pour la découverte et la localisation de ressources, le transfert de données et le déploiement d'applications. Cependant, la conception d'applications pour les grilles de calcul reste complexe.

Même si des solutions ont été apportées pour les transferts de données (GridFTP [1]) ou leur stockage (*Internet Backplane Protocol*, IBP [11]). Nous avons vu au chapitre 2 que les solutions existantes ne permettaient pas d'offrir aux applications un service fournissant un accès transparent aux données partagées en garantissant à la fois leur *persistance* et leur *cohérence*.

Contributions

Notre travail se situe dans le cadre du service de partage de données JUXMEM. Ce service offre aux applications la possibilité de partager des données de manière transparente. Il gère de manière transparente le *stockage persistant*, la *localisation*, le *transfert* et la *cohérence* des données. La conception du service de partage de données JUXMEM s'inspire à la fois de solutions proposées 1) dans le cadre des systèmes pair-à-pair (P2P) pour la localisation des ressources et des données ; 2) dans le cadre des systèmes à mémoire virtuellement partagée (MVP) pour les modèles et protocoles de cohérence ; et 3) dans le cadre des systèmes distribués tolérants aux fautes pour la conception de nos mécanismes garantissant la persistance des données.

Le service de partage de données JUXMEM a fait l'objet d'un dépôt à l'Agence pour la Protection des Programmes (APP) et est disponible en téléchargement [129].

Au sein de ce service, nous nous somme focalisés sur la gestion de la *cohérence des données* et le support pour la *tolérance aux fautes*. Nous proposons une architecture permettant une gestion *conjointe* de la cohérence des données et de la tolérance aux fautes. Les solutions proposées suivent une approche hiérarchique afin de s'adapter aux grilles de calcul. La section 9.2.2 montre les bénéfices qu'offre une telle approche.

Gestion conjointe de la cohérence des données et la tolérance aux fautes. Nous proposons une architecture logicielle en couches permettant de *découpler* la gestion de la cohérence des données de celle de la tolérance aux fautes. Des interfaces générales ont été mises en place entre ces couches.

Nous avons montré au chapitre 8 que cette approche permet de concevoir des protocoles de cohérence sans avoir à prendre en compte les mécanismes de tolérance aux fautes sous-jacents. Il est ainsi possible d'utiliser différentes mises en œuvre des

couches de tolérance aux fautes avec une même mise en œuvre de protocole de cohérence[1].

Pour chaque donnée partagée, il est possible de choisir la mise en œuvre de la couche de tolérance aux fautes ainsi que le protocole de cohérence utilisé. Le choix s'effectue à l'allocation de chaque donnée. Cet aspect multi-protocole permet aux applications de choisir la combinaison la mieux adaptée pour chaque donnée. Nous avons illustré à la section 9.2.3 l'intérêt d'offrir une telle possibilité dans le cas particulier de la visualisation d'une donnée partagée.

Approche hiérarchique pour la gestion de la cohérence. De nombreux protocoles de cohérence sont basés sur la notion de copie de référence ou utilisent un nœud jouant un rôle particulier, comme un nœud gestionnaire d'une donnée, chargé de localiser la copie à jour. Ces protocoles sont mal adaptés à la topologie réseau hiérarchique intraorganisation/interorganisation des grilles de calcul : si des clients accèdent à une donnée dont la copie de référence est située au sein d'une organisation différente, les liens interorganisation vont être utilisés pour chaque accès à la copie de référence. Notre approche consiste à placer un représentant local de la copie de référence d'une donnée au sein de chaque organisation dans laquelle se trouvent des clients accédant à cette donnée. Ces représentants locaux, appelés *copies de référence locales*, agissent comme des caches et permettent de réduire l'utilisation des liens interorganisation. La présence des copies de référence locales apporte des bénéfices significatifs lors des accès aux données (section 9.2.2).

Approche hiérarchique pour la gestion de la tolérance aux fautes. Les copies de référence locales et globales des protocoles de cohérence représentent des entités critiques qu'il semble judicieux de préserver. Nous proposons des *groupes auto-organisants hiérarchiques* : chaque copie de référence locale au sein d'une organisation est répliquée sur un groupe de fournisseurs appelé *Local Data Group* (LDG) ; la copie de référence globale est répliquée dans un groupe dont les membres sont les copies de référence locales, ce groupe est appelé *Global Data Group* (GDG). Chaque donnée se voit ainsi associer un GDG comprenant plusieurs LDG eux-mêmes composés de nœuds fournisseurs hébergeant une copie de la donnée.

En annexe, nous présentons une approche probabiliste visant une plus grande échelle ainsi qu'une plus forte dynamicité. Cette approche met en œuvre un réseau logique pair-pair malléable.

Réseau logique pair-à-pair malléable. Nous nous sommes intéressé à la réplication dans des environnements de type pair-à-pair. Les mécanismes de réplication nécessitent l'utilisation de communication de groupe, notamment la diffusion. Dans les systèmes pair-à-pair, les communications suivent les chemins entre les pair voisins dans le réseau logique. Le réseau logique que nous proposons est malléable, c'est-à-dire qu'il rapproche les pairs appartenant à un même groupe. Cela permet d'améliorer les communications entre ces pairs et par conséquent d'améliorer les diffusions au sein des groupes.

[1] Ceci suppose cependant que la mise en œuvre de la couche de tolérance aux fautes offre des propriétés de diffusions atomiques.

Perspectives

Les travaux que nous avons présentés dans ce manuscrit débouchent sur des perspectives présentées ci-dessous.

Équilibrage de charge

Dans la version actuelle, les protocoles de cohérence client accèdent aux données par l'intermédiaire de la copie de référence locale. Les requêtes modifiant l'état de cette copie de référence sont diffusées sur chaque nœud du LDG sur lequel cette copie est répliquée. Les requêtes ne modifiant pas l'état sont transmises à un seul membre de ce groupe. De plus, de manière à masquer la réplication du côté client, nous avons choisi que les clients ne connaissent qu'un représentant du groupe. En cas de défaillance de celui-ci, un nouveau représentant est recherché.

Une nouvelle approche pourrait consister à permettre au client de connaître tous les membres du groupe (LDG) ou seulement un sous-ensemble. Il deviendrait ainsi possible de mettre en place des politiques d'équilibrage de charge lors des requêtes ne modifiant pas l'état de la copie de référence locale. Plusieurs clients pourraient par exemple récupérer une copie de la donnée en parallèle sur des nœuds fournisseurs différents. Grâce à cette approche, les copies mises en place par les mécanismes de réplication à des fins de tolérance aux fautes pourraient être exploitées pour améliorer les performances des lectures.

Utilisation de codes correcteurs

Les mécanismes de tolérance aux fautes présentés dans ce manuscrit répliquent chaque donnée dans son intégralité sur plusieurs nœuds fournisseurs afin de pourvoir supporter la perte de certains d'entre eux. Ces mécanismes sont cependant coûteux en terme d'espace de stockage, d'autant plus pour les données de grande taille.

Nous n'avons pas exploré des mécanismes de tolérance aux fautes utilisant des codes correcteurs. Un code correcteur est une technique de codage fondée la redondance d'information et utilisée afin de pouvoir corriger des défaillances des transmission de données mais aussi des supports de stockage. Au sein d'un service de partage de données pour la grille, cela consisterait à découper les données en blocs, à calculer des blocs de redondance et à répartir les différents blocs sur plusieurs fournisseurs.

Les codes correcteurs peuvent ainsi offrir un bon équilibrage de charge, notamment lors des lectures. Cependant, chaque écriture implique le calcul des nouveaux blocs de redondance. Il serait intéressant de mener une étude comparative de ces deux types d'approche.

Notons que l'utilisation de codes correcteurs pour le stockage de la donnée n'est pas incompatible avec l'utilisation de nos mécanismes de réplication qui serviraient alors à répliquer l'état du protocole de cohérence (identité du détenteur du verrou, listes d'attentes, etc.) alors que les codes correcteurs pourraient être utilisés pour le stockage de la donnée proprement dite.

Tolérance aux fautes adaptative

Actuellement, les clients du service de partage de données JUXMEM doivent préciser eux-mêmes les mécanismes de tolérance aux fautes à utiliser, ainsi que le degré de réplication. De plus, la couche d'adaptation aux fautes ne possède qu'une politique : remplacer les fournisseurs défaillants. Le degré de réplication d'une donnée ne varie donc jamais.

Le type de mécanisme de tolérance aux fautes à utiliser ainsi que le degré de réplication devraient pouvoir être déterminés en fonction du niveau de criticité de la donnée et du degré de risque. L'application pourrait alors fournir un niveau de criticité de la donnée en fonction des garanties souhaitées en terme de tolérance aux fautes. Par exemple le niveau de criticité pourrait être : "très critique, ne doit pas être perdue", "critique", "non critique", etc. Le degré de risque peut être estimé par un module *d'observation* du système, par exemple en prenant en compte les fautes survenues récemment, la température des machines, la charge réseau, etc. Ce degré de risque étant susceptible de changer au cours du temps, la couche d'adaptation aux fautes pourrait dans certains cas décider de ne pas remplacer un nœud défaillants, ou au contraire de le remplacer par plusieurs nœuds en prévision de pannes futures.

Défaillances des clients

Dans notre travail, nous nous sommes focalisés sur les fautes du service de partage de données, c'est-à-dire sur les défaillances des nœuds hébergeant les données, les nœuds fournisseurs. Pour tolérer les défaillances des nœuds clients, nous avons proposé de mettre en œuvre des mécanismes de points de reprise hiérarchiques. L'intégration des deux solutions n'est pas triviale. Si les applications utilisant le service de partage de données sont susceptibles d'effectuer des retours en arrière au niveau d'un point de reprise sauvegardé, le service doit être en mesure de restaurer l'état du protocole de cohérence ainsi que la version de la donnée qui correspond au moment de la sauvegarde du point de reprise. Ceci implique que le service conserve plusieurs versions de la donnée et soit informé des sauvegardes de points de reprise.

Le service de partage de donnée peut en revanche tirer avantage de la possibilité de retour en arrière des applications : des mécanismes de réplication optimistes peuvent être mis en place et le service peut alors forcer un retour en arrière de l'application en cas de perte de la dernière version de la donnée.

Vers les fédérations de grilles

De nombreux projets de grilles sont réalisés à travers le monde. De plus en plus d'initiatives visant à relier des grilles entre elles voient le jour. On peut donc penser qu'il existera, d'ici quelques années, des *fédérations de grilles*. Notre approche hiérarchique pour la gestion de la cohérence des données et de la tolérance aux fautes a été conçue pour une hiérarchie à deux niveaux : intraorganisation et interorganisation. Il semble raisonnable de penser que deux grilles seront interconnectées, ajoutant ainsi un niveau de hiérarchie dans la topologie réseau. Une approche basée sur une hiérarchie à deux niveaux sera alors insuffisante. Ce cas peut également se présenter au sein d'une grille si l'une des organisations est composée de deux sous-organisations possédant chacune une grappe de calculateurs. Il semble

intéressant de généraliser notre approche hiérarchique à deux niveaux en une approche hiérarchique à n niveaux.

Références

Les premières références bibliographiques correspondent à nos propres publications. Elles sont listées à la page 3.

[1] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[2] Yair Amir and Ciprian Tutu. From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, page 494, Washington, DC, 2002. IEEE Computer Society.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem : Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service. *Kluwer Journal of Supercomputing*, 2005. To appear. Preliminary electronic version available as INRIA Research Report RR-5082.

[5] Gabriel Antoniu, Luc Bougé, and Sébastien Lacour. Making a DSM consistency protocol hierarchy-aware: An efficient synchronization scheme. In *Proceedings of the 3rd IEEE/ACM International Conference on Cluster Computing and the Grid (CCGrid '03)*, pages 516–523, Tokyo, Japan, May 2003. IEEE.

[6] Luciana Bezerra Arantes, Pierre Sens, and Bertil Folliot. An Effective Logical Cache for a Clustered LRC-Based DSM System. *Cluster Computing Journal*, 5(1):19–31, January 2002.

[7] Vadim Astakhov, Amarnath Gupta, Simone Santini, and Jeffrey S. Grethe. Data Integration in the Biomedical Informatics Research Network (BIRN). In *Data Integration in the Life Sciences (DILS '05)*, pages 317–320, 2005.

[8] Morin Christine Badrinath Ramamurthy. Common Mechanisms for Supporting Fault Tolerance in DSM and Message Passing Systems. Research Report RR-4613, INRIA, IRISA, Rennes, France, November 2002.

[9] R. Baldoni, J. M. Hélary, A. Mostefaoui, and M. Raynal. Consistent State Restoration in Shared Memory Systems. In *Proc. of the Int. IEEE Conference on Advances in Parallel and Distributed Computing (APDC '97)*, pages 330–337, Shangai, 1997.

[10] A.-L. Barabasi. *LINKED: The New Science of Networks*. Perseus Books Group, 2002.

[11] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A study in resource sharing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '02)*, pages 194–201, Berlin, Germany, May 2002. IEEE.

[12] Micah Beck, Ying Ding, Terry Moore, and James S. Plank. Transnet Architecture and Logistical Networking for Distributed Storage. In *Workshop on Scalable File Systems and Storage Technologies (SFSST)*, San Francisco, CA, September 2004. Held in conjunction with the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004).

[13] Micah Beck, Terry Moore, and James S. Plank. An end-to-end approach to globally scalable network storage. In *Proceedings of the 2002 Conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '02)*, pages 339–346. ACM Press, 2002.

[14] Micah Beck, Terry Moore, James S. Plank, and Martin Swany. Logistical Networking: Sharing More Than the Wires. In *Proceedings of 2nd Annual Workshop on Active Middleware Services*, volume 583 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, August 2000.

[15] Brian N. Bershad, Mattew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON '93)*, pages 528–537, Los Alamitos, CA, February 1993.

[16] Marin Bertier. *Service de détection de défaillances hiérarchique*. Thèse de doctorat, Université de Paris 6, LIP6, Paris, France, 2004.

[17] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*, pages 354–363, Washington, DC, June 2002.

[18] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, pages 635–644, San Francisco, CA, june 2003.

[19] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: a data movement and access service for wide area computing systems. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems (IOPADS '99)*, pages 78–88, New York, NY, may 1999. ACM Press.

[20] Kenneth Birman and Robert Cooper. The ISIS project: real experience with a fault tolerant programming system. *SIGOPS Oper. Syst. Rev.*, 25(2):103–107, 1991.

[21] Bela Bollobas. *Random Graphs, Second Edition*. Cambride University Press, United Kingdom, 2001.

[22] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC '02)*, pages 1–18, Baltimore, Maryland, November 2002. IEEE Computer Society.

[23] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis: A Highly-Scalable Multi-user Peer-to-Peer File System. In *Proceedings of the 11th International Euro-Par Conference (Euro-Par '05)*, volume 3648 of *Lecture Notes in Computer Science*, pages 1173–1182, Lisbon, Portugal, August 2005. Springer.

[24] Sidney Cadot, Frits Kuijlman, Koen Langendoen, Kees van Reeuwijk, and Henk Sips. ENSEMBLE: A communication layer for embedded multi-processor systems. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES '01)*, pages 56–63, New York, NY, 2001. ACM Press.

[25] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[26] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Emmanuel Jeannot, Yvon Jegou, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05)*, Seattle, Washington, November 2005.

[27] Gianni Di Caro and Marco Dorigo. AntNet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[28] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[29] Mark Carson and Darrin Santay. Nist net: a Linux-based network emulation tool. *SIGCOMM Computing Communication Review*, 33(3):111–126, 2003.

[30] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation: (OSDI '99)*, volume 32(5) of *Operating Systems Review*, pages 173–186, New Orleans, LA, February 1999. Usenix Association.

[31] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[32] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[33] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Computer Systems*, 3(1):63–75, February 1985.

[34] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. The Legion Resource Management System. In *5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99)*, volume 1659 of *Lecture Notes in Computer Science*, pages 162–178, San Juan, Puerto Rico, April 1999. Springer.

[35] Shu-Wie F. Chen and Calton Pu. An analysis of replica control. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, pages 22–25, Monterey, CA, November 1992.

[36] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

[37] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[38] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, number 2009, pages 46–66, Berkeley, CA, July 2000.

[39] Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, and Miguel Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proceedings of the Second Symposium on Operating Systems Design and Implementations (OSDI '96)*, pages 59–73, Seattle, WA, October 1996.

[40] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[41] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*, Washington, DC, June 2002.

[42] B. Devianov and S. Toueg. Failure detector service for dependable computing. In *Proceedings of the First International Conference on Dependable Systems and Networks*, pages 14–15, New York, NY, Juin 2000.

[43] S. G. Djorgovski, R. R. Gal, S. C. Odewahn, R. R. de Carvalho, R. Brunner, G. Longo, and R. Scaramella. The Palomar Digital Sky Survey (DPOSS), 1998. http://astro.ncsa.uiuc.edu/catalogs/dposs/.

[44] F. Dunno, L. Gaido, A. Gishelli, F. Prelz, and M. Sgaravatoo. DataGrid Prototype 1. EU-DataGrid Collaboration. In *Proceedings of the TERENA Networking Conference*, Limerick, Ireland, June 2002.

[45] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[46] Xavier Défago, André Schiper, and Péter Urban. Totally ordered broadcast and multicast algorithms: a comprehensive survey. Technical Report TR DSC/2000/036, Dept. of Communication Systems, EPFL, Switzerland, October 2000.

[47] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[48] P. Erdös and A. Rényi. On the evolution of random graphs. *Publication of the Mathemathic Institue of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[49] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[50] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and DHTs. In *Proceedings 2nd Workshop on Real, Large, Distributed Systems (WORLDS '05)*, pages 55–60, San Francisco, CA, December 2005.

[51] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA '90)*, pages 15–26, Seattle, WA, June 1990.

[52] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD '96)*, pages 173–182, Montreal, Canada, 1996. ACM Press.

[53] F. Greve, M. Hurfin, M. Raynal, and F. Tronel. Primary component asynchronous group membership as an instance of a generic agreement framework. In *Proceedings of the IEEE International Symposium on Autonomous Decentralized Systems (ISADS '01)*, pages 93–100, Dallas, Texas, March 2001.

[54] Fabíola Gonçalves Pereira Greve. *Réponses efficaces au besoin d'accord dans un groupe*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, Novembre 2002.

[55] Network Working Group. RFC 2988: Computing TCP's retransmission, 2000.

[56] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[57] Sidath Handurukande, Anne-Marie Kermarrec, Fabrice Le Fessant, and Laurent Massoulié. Exploiting semantic clustering in the eDonkey P2P network. In *Proceedings of SIGOPS European Workshop*, pages 109–114, Leuven, Belgium, September 2004.

[58] William Hoarau and Sébastien Tixeuil. A language-driven tool for fault injection in distributed system. Research report 1399, Laboratoire de Recherche en Informatique (LRI), University Paris-Sud, France, February 2005.

[59] Jeff Hodges and Robert Morgan. Lightweight Directory Access Protocol (v3): Technical Specification. IETF Request For Comment 3377, Network Working Group, 2002.

[60] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS '90)*, pages 302–311, May.

[61] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 277–287, Padova, Italy, June 1996.

[62] Van Jacobson. Congestion avoidance and control. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '88)*, pages 314–329, Stanford, CA, August 1988.

[63] Neel K. Jain. Group formation mechanisms for transactions in ISIS. In *Proceedings of the third international conference on Information and knowledge management (CIKM '94)*, pages 203–210, New York, NY, 1994. ACM Press.

[64] Mathieu Jan. *JuxMem: un service de partage transparent de données pour grilles de calculs fondé sur une approche pair-à-pair*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, November 2006.

[65] Màrk Jelasity and Ozalp Babaoglu. T-Man: Fast Gossip-based Contruction of Large-Scale Overlay Topologies. Technical Report UBLCS-2004-7, University of Bologna, May 2004.

[66] Mark Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware '04)*, pages 79–98, New York, NY, 2004. Springer.

[67] Yvon Jégou. Implementation of Page Management in Mome, a User-Level DSM. In *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid '03)*, pages 479–486, Tokyo, Japan, May 2003. IEEE Computer Society.

[68] Anne-Marie Kermarrec, Gilbert Cabillic, Alain Gefflaut, Christine Morin, and Isabelle Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems (FTCS '95)*, pages 289–298, Pasadena, CA, June 1995.

[69] Anne-Marie Kermarrec and Christine Morin. Smooth and efficient integration of high-availability in a parallel single level store system. In *Proceedings of the 7th International Euro-Par Conference on Parallel Processing (Euro-Par '01)*, volume 2150 of *Lecture Notes in Computer Science*, pages 752–763, Manchester, UK, 2001. Springer.

[70] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[71] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[72] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Ken Birman, and Al Demers. Decentralized schemes for size estimation in large and dynamic groups. In *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications (NCA '05)*, pages 41–48, Washington, DC, July 2005. IEEE Computer Society.

[73] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 2000)*, volume 2218 of *Lecture Notes in Computer Science*, pages 190–201, Cambridge, MA, 2000. Springer.

[74] Sébastien Lacour. *Contribution à l'automatisation du déploiement d'applications sur les grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2005.

[75] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[76] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[77] J.-C. Laprie. Dependable computing and fault tolerance : concepts and terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS '85)*, pages 2–11, Ann Arbor, MI, June 1985.

[78] Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of Distributed Computing, 13th International Symposium 1999 (DISC '99)*, volume 1693 of *Lecture Notes in Computer Science*, pages 34–48, Bratislava, Slavak Republic, September 1999. Springer.

[79] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[80] D. Manivannan and Mukesh Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transaction on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[81] Olivier Marin, Marin Bertier, and Pierre Sens. DARX - A Framework for the Fault-Tolerant Support of Agent Software. In *14th IEEE International Symposium on Software Reliability Engineering (ISSRE '03)*, page 406, Denver, CO, November 2003.

[82] Sergio Mena, André Schiper, and Pawel Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of the 4th International Middleware Conference (Middleware '03)*, volume 2672 of *Lecture Notes in Computer Science*, pages 414–432, Rio de Janeiro, Brazil, June 2003. Springer.

[83] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. *IEEE Transaction on Parallel and Distributed Systems*, 8(9):959–969, September 1997.

[84] Christine Morin, Anne-Marie Kermarrec, Michel Banâtre, and A. Gefflaut. An efficient and scalable approach for implementing fault tolerant DSM architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.

[85] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, 2002.

[86] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Gnutella, pages 94–122. O'Reilly, May 2001.

[87] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the 25th International Conference on Very Large Data Base (VLDB '99)*, pages 126–137, San Francisco, CA, 1999. Morgan Kaufmann Publishers Inc.

[88] Cécile Le Pape. *Contrôle de Qualité des Données Répliquées dans un Cluster*. Thèse de doctorat, Université Pierre et Marie Curie, LIP6, Paris, France, December 2005.

[89] James Plank, Micah Beck, Wael Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: Storage in the network. In *Network Storage Symposium (NetStore '99)*, Seattle, WA, October 1999.

[90] Jelica Protić, Milo Tomasević, and Veljko Milutinović. *Distributed Shared Memory: Concepts and Systems*. IEEE, August 1997.

[91] A. Rajasekar, M. Wan, R.W. Moore, W. Schroeder, G. Kremenek, A. Jagatheesan, C. Cowart, B. Zhu, S.Y. Chen, and R. Olschanowsky. Storage resource broker - managing distributed data in a grid. *Computer Society of India Journal*, 33(4):42–54, 2003. special issue on SAN.

[92] Michel Raynal, Matthieu Roy, and Ciprian Tutu. Merging atomic consistency and sequential consistency. Technical Report 1629, IRISA, June 2004.

[93] J. Rough and A. Goscinski. Exploiting operating system services to efficiently checkpoint parallel applications in GENESIS. In *Proceedings of the 5th IEEE International Conference on Algorithms and Architectures for Parallel Processing*, pages 261–268, October 2002.

[94] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–250, Heidelberg, Germany, November 2001. Springer.

[95] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *In Proceedings of Networked Group Communication (NGC '01)*, pages 30–43, London, UK, 2001.

[96] Ian Foster S. Vazhkudai, Steven Tuecke. Replica selection in the Globus data grid. In *1st IEEE/ACM International Conference on Cluster Computing and the Grid (CCGrid '01)*, pages 106–113, Brisbane, Australia, May 2001. IEEE.

[97] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, January 2002.

[98] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[99] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '01)*, pages 149–160, San Diego, CA, August 2001.

[100] F. Sultan, T. D. Nguyen, and L. Iftode. Lazy garbage collection of recovery state for fault-tolerant distributed shared memory. 13(7):673–686, July 2002.

[101] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[102] Spyros Voulgaris, Daniela Gavida, and Maarten Van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2), June 2005.

[103] Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proceedings of the 11th International Euro-Par Conference (Euro-Par 2005)*, volume 3648 of *Lecture Notes in Computer Science*, pages 1143–1152, Lisboa, Portugal, August 2005. Springer.

[104] D. J. Watts and S. H. Strogatz. Collective dynamics of small world networks. *Nature*, 393:440–442, 1998.

[105] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (Supercomputing '01)*, page 59, New York, NY, 2001. ACM Press.

[106] Ben Yanbin Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division (EECS), Univeristy of California, Berkeley, CA, April 2001.

[107] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems (OSDI '96). In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 75–88, Seattle, WA, October 1996.

[108] eDonkey. http://www.edonkey2000.com/.

[109] The EUROGRID project. http://www.eurogrid.org/.

[110] The general atomic and molecular electronic structure system (GAMESS). http://www.msg.ameslab.gov/GAMESS/GAMESS.html.

[111] The GDS project: A Grid Data Service. http://www.irisa.fr/GDS/.

[112] The Globus project. http://www.globus.org/.

[113] Grid'5000 Project. http://www.grid5000.org/.

[114] JXTA Distributed Framework. http://jdf.jxta.org/, 2003.

[115] JXTA specification project. http://spec.jxta.org/.

[116] KaZaA. http://www.kazaa.com/.

[117] Legion - the worldwide virtual computer. http://www.cs.virginia.edu/~legion/.

[118] MCAT - "the metadata catalog". http://www.npaci.caltech.edu/2mass/.

[119] Myrinet performance measurements. http://www.myrinet.com/myrinet/performance/.

[120] MySQL. http://www.mysql.com/.

[121] OAR. http://oar.imag.fr/.

[122] OARGRID. http://oargrid.gforge.inria.fr/.

[123] The PARIS research group: Programming distributed parallel systems for large scale numerical simulation. http://www.inria.fr/recherche/equipes/paris.en.html.

[124] PlanetLab website. http://www.planet-lab.org/.

[125] Renater : Le Réseau National de Télécommunications pour la Technologie, l'Enseignement et la Recherche. http://www.renater.fr/.

[126] The TeraGrid project. http://www.teragrid.org/.

[127] The TeraShake project. http://sceclib.sdsc.edu/TeraShake/.

[128] Le projet HydroGrid. http://www-rocq.inria.fr/~kern/HydroGrid/HydroGrid.html.

[129] The JuxMem Project: Juxtaposed Memory. http://juxmem.gforge.inria.fr/.

Cinquième partie

Annexes

Nous présentons ici un travail effectué dans le cadre d'une collaboration avec Indranil Gupta et Ramsés Morales de l'université de l'Illinois à Urbana-Champaign (*University of Illinois at Urbana-Champaign* ou UIUC) aux États-Unis. Dans le cadre de cette collaboration, j'ai effectué un séjour d'un mois à Urbana et nous avons reçu à Rennes Ramsés Morales durant un mois ainsi que Indranil Gupta durant 15 jours. Le travail présenté ici sort donc du cadre du service de partage de données pour la grille JUXMEM.

Jusque-là, nous nous sommes intéressé à la gestion de la tolérance aux fautes et de la cohérence des données dans un cadre très précis (JUXMEM) pour une architecture physique de type grille et des applications scientifiques. Ceci implique de nombreuses contraintes, comme la présence d'un protocole de cohérence de données offrant des garanties de cohérence aux applications scientifiques. Il existe des cadres dans lesquels les besoins des applications en terme de garantie et de performance ne sont pas aussi forts. C'est notamment le cas des applications pair-à-pair collaboratives. En revanche, pour ces dernières, supporter la volatilité des nœuds est crucial, les nœuds n'étant plus forcément des calculateurs dédiés mais des ordinateurs personnels par exemple. Nous avons donc étudié comment offrir à ce type d'application la possibilité d'avoir des mécanismes de communication de groupe efficaces à partir du réseau logique pair-à-pair.

Les réseaux logiques pair-à-pair permettent aux applications distribuées de s'exécuter à grande échelle de manière complètement distribuée et tolérante aux fautes. Cependant la plupart des réseaux logiques présentés dans la littérature, structurés et non-structurés, sont inflexibles du point de vue de l'application. En d'autres termes, l'application n'a pas de contrôle sur la structure du réseau logique qu'elle utilise. Nous proposons le concept d'un réseau logique malléable et la conception du premier réseau logique malléable que nous appelons MOVE (pour l'anglais *Malleable Overlay*). Les caractéristiques des schémas de communications d'une application distribuée utilisant MOVE peuvent influencer la structure même du réseau logique. Ceci avec un double but : 1) optimiser les performances des applications en adaptant le réseau logique, tout en 2) conservant les propriétés de passage à l'échelle et de tolérance aux fautes des réseaux logiques pair-à-pair. Cette influence peut être spécifiée explicitement par l'application ou glanée par nos algorithmes. En plus de gérer des listes de voisins (constituant le réseau) MOVE propose des algorithmes de découverte de ressources, de propagation des mises à jour et de résistance à la volatilité. Quand l'application a peu de besoins de communication, la plupart des liens du réseau logique conservent leur configuration par défaut ; par contre, quand les schémas de communication de l'application deviennent plus exigeants, le réseau logique s'adapte de lui-même en favorisant les liens entre les entités communicantes.

A.1 Prise en compte des applications au niveau d'un réseau logique pair-à-pair

A.1.1 Un réseau logique malléable

Il existe deux grandes familles de réseaux logiques pair-à-pair : les réseaux dits *structurés*, fondés sur des *tables de hachage distribuées*, comme Pastry et Chord [94, 99] et ceux dits *non-structurés*, construits sur des protocoles épidémiques ou d'inondation, comme Freenet, Gnutella, KaZaA [38, 86, 116]. Ces réseaux logiques pair-à-pair sont bien adaptés à la vola-

tilité, arrivées et départs de nœuds, et supportent également les grandes échelles de l'ordre de millions de nœuds.

Cependant, les réseaux logiques structurés ou non structurés, présentent l'inconvénient d'être *inflexibles* du point de vue des applications. Les règles et invariants qui servent au choix et au maintien des nœuds voisins dans le réseau logique sont dictés de manière rigide : par exemple le choix des voisins est souvent guidé par le résultat d'une fonction de hachage. Ceci implique que les développeurs d'applications distribuées pair-à-pair doivent soit se contenter du réseau logique proposé, soit en concevoir un dédié à leur application.

Afin de répondre à ce problème, nous proposons le concept de réseau logique malléable.

Définition A.1 : ***réseau logique malléable.*** — Un réseau logique malléable est défini comme un réseau logique pair-à-pair au sein duquel les caractéristiques des applications distribuées l'utilisant peuvent *influencer* la structure même du réseau logique.

Le double but poursuivi par un réseau logique malléable est : 1) optimiser les performances des applications utilisatrices à partir du réseau logique, tout en 2) conservant les propriétés de passage à l'échelle et de tolérance à la volatilité de l'approche *réseau logique pair-à-pair*.

Afin de réaliser et d'évaluer le concept de réseau logique malléable, nous avons construit un réseau logique malléable spécifique que nous avons appelé MOVE. MOVE combine des éléments d'un réseau logique *non structuré* avec les caractéristiques des applications. La structure et le comportement du réseau sont influencés à la fois par le réseau logique non structuré sous-jacent et par les caractéristiques des applications. Cette influence peut être 1) explicitement spécifiée par les applications, ou 2) due aux caractéristiques détectées automatiquement par nos algorithmes.

Dans les réseaux logiques pair-à-pair, chaque nœud maintient une liste de nœuds voisins. L'ensemble de ces listes détermine la relation *qui connaît qui*. Les listes de nœuds voisins ne contiennent pas tous les nœuds du système mais seulement quelques-uns [41, 27, 99]. MOVE propose des algorithmes pour la maintenance des listes de voisins, la propagation de mises à jour, et la résistance à la volatilité. Lorsque les applications ont peu de communications, le réseau logique maintenu par MOVE ressemble à un réseau logique non-structuré classique. Par contre, au fur et à mesure que les caractéristiques des applications sont détectables, il s'adapte à l'application tout en conservant une partie de sa structure par défaut.

Les applications pair-à-pair collaboratives, comme un tableau blanc distribué, un service de conférence audio et vidéo, un service de gestion de données répliquées ou une plateforme de jeux distribuée sont des applications motivantes pour un réseau logique pair-à-pair comme MOVE. En effet, elles présentent le point commun de reposer sur une notion de *groupe applicatif*. Chaque processus applicatif appartient à un ou plusieurs groupes et interagit avec les autres processus des mêmes groupes. Par exemple, les membres d'un même groupe peuvent partager un état qui doit être mis à jour (le tableau blanc, le tableau de jeu, les copies d'une donnée répliquée, etc.). Prendre en compte cette notion de groupe au niveau du réseau logique permet de faire profiter toute application de cette optimisation sans qu'elle ait à le prendre en compte de manière spécifique.

MOVE permet à de telles applications reposant sur la notion de groupe d'influencer le réseau logique qui peut être utilisé conjointement par de multiples applications collaboratives. Le but du réseau logique MOVE est triple.

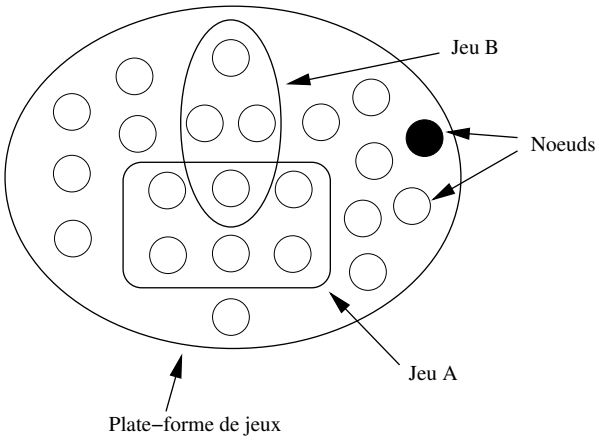

FIG. A.1 – Une plate-forme de jeux collaborative.

La connectivité. Le diamètre du graphe du réseau logique doit rester petit afin de permettre aux requêtes d'être propagées rapidement.

Mises à jour. Les mises à jour au sein des groupes applicatifs doivent être efficaces.

Tolérance à la volatilité. Le réseau doit conserver ses propriétés en présence de volatilité.

L'idée de départ de notre approche est de maintenir sur chaque nœud des listes de voisins qui par défaut ne contiennent que des *liens non-applicatifs*, c'est-à-dire des voisins choisis aléatoirement. Ensuite, lorsque des groupes applicatifs se forment, certains des voisins non-applicatifs sont automatiquement remplacés par des voisins qui se situent dans le même groupe applicatif. Un voisin non-applicatif peut soit *devenir* un voisin applicatif si les voisins appartiennent au même groupe ou *être remplacé* par un voisin applicatif.

A.1.2 Exemples de scénarios

Considérons la plate-forme de jeux collaborative à grande échelle représentée par la figure A.1 comme exemple d'application. Des dizaines de milliers de nœuds répartis sur tout Internet peuvent participer à cette application. Pour des raisons de performances la taille des listes de voisins sur chaque nœud doit être bornée. En effet, chaque entrée dans ces listes correspond à un voisin qui doit être surveillé et dont l'état doit être maintenu à jour. Par conséquence, chaque nœud n'a qu'une vision partielle de la plate-forme complète. Ceci ne doit pas avoir un impact négatif sur les propriétés recherchées par l'application comme la connectivité, l'efficacité de la propagation des mises à jour et la tolérance à la volatilité.

Connectivité. Un réseau logique est dit *connecté* s'il existe un chemin (succession d'arêtes ou liens) entre chaque paire de nœuds. Cette propriété est très importante car elle apporte la garantie que tout nœud peut communiquer avec tout autre nœud dans le réseau.

Un nœud particulier, par exemple le noir sur la figure A.1, peut avoir à rechercher une partie de jeu spécifique au sein de la plate-forme, par exemple le jeu A. Seulement un sous-ensemble des nœuds participe à ce jeu, disons quelques dizaines. Il est tout à fait possible qu'aucun des voisins contenus dans la liste de voisins du nœud noir ne participe à cette

partie-là. Il faut que l'ensemble de la plate-forme de jeu soit connectée afin de donner un moyen à tout nœud d'atteindre n'importe quel autre nœud, éventuellement via un long chemin. La recherche d'un nœud participant au jeu A dépend de l'application : dans la pratique, cela peut consister à visiter une page sur Internet, à consulter un outil de recherche spécialisé ou encore a envoyer une recherche de type *inondation* à travers le réseau logique.

À des fins de performance, le diamètre du réseau doit être aussi petit que possible même si les listes de voisins sont partielles. Notons cependant qu'il suffit à un nouveau joueur de localiser *un* des joueurs d'une partie pour être en mesure de joindre tous les autres nœuds de la partie.

Mises à jour efficaces. Au cours d'une partie de jeu, les joueurs possèdent chacun la réplique d'un objet représentant l'état courant de la partie : selon l'application, cela peut être un tableau blanc, une donnée répliquée, etc. Chaque fois qu'un joueur joue, il met à jour sa réplique de l'objet. Afin que les autres joueurs puissent jouer, ils doivent voir les modifications apportées par le joueur qui vient de jouer. Pour cela, les mises à jour doivent être propagées de manière efficace au sein d'un groupe applicatif, par exemple le jeu A ou le jeu B sur la figure A.1. Au niveau du réseau logique, le moyen d'optimiser la vitesse de propagation de ces mises à jour est de limiter le nombre de sauts dans le réseau logique entre deux nœuds appartenant au même groupe applicatif.

Tolérance à la volatilité. Parmi plusieurs milliers de nœuds répartis à travers Internet, il est très probable que quelques-uns subissent une faute ou se retrouvent déconnectés [97]. Au niveau de la plate-forme entière, cela ne doit pas mener à rompre la connectivité du réseau logique.

Au niveau des groupes applicatifs, de tels événements ne doivent pas stopper les communications : les nœuds restants doivent restés connectés entre eux. D'autre part, le départ soudain d'un nœud peut supprimer quelques chemins dans le graphe du groupe applicatif ce qui implique que le diamètre du graphe du groupe risque d'augmenter. Cependant, les mécanismes de propagation des mises à jours doivent rester efficaces.

A.2 Conception d'un réseau logique pair-à-pair malléable

Le premier but d'un réseau logique est de connecter entre eux les différents nœuds. La première propriété à remplir est donc la *connectivité* du réseau. De plus, nous avons souligné, dans la section précédente, l'importance de fournir un support pour des propagations de mise à jour efficaces au sein des groupes applicatifs. Ceci peut être favorisé en introduisant un peu de regroupement (*clustering*) au sein du réseau logique. Il est nécessaire de préserver à la fois la connectivité et le regroupement lorsque l'on prend en compte la nature dynamique de l'environnement.

Propriétés des graphes aléatoires. Les *graphes aléatoires* présentent de bonnes propriétés en terme de connectivité et de distribution de degrés [48, 10]. Par exemple, si chaque sommet d'un graphe aléatoire de taille N (N grand) a au moins $\log(N)$ sommets voisins choisis aléatoirement (selon une distribution uniforme) alors le graphe aléatoire aura une très grande

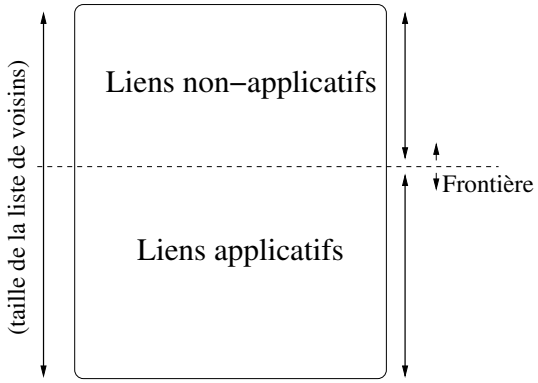

FIG. A.2 – La liste de voisins sur chaque nœud.

probabilité d'être connexe [21]. L'estimation de la taille (N) d'un système distribué, à grande échelle, dynamique a fait l'objet de recherches [72]. Cependant, nous visons des échelles de quelques milliers à quelques dizaines de milliers de nœuds, aussi pouvons-nous prendre une borne supérieure : 50 liens par nœuds, par exemple, laissent une grande marge de sécurité pour connecter théoriquement jusqu'à 5×10^{21} nœuds. Les graphes aléatoires présentent également une bonne distribution des degrés. Un réseau logique basé sur un graphe aléatoire peut bénéficier de cela pour offrir une bonne *distribution de charge*. Pour toutes ces raisons, les algorithmes de MOVE essayent de conserver une partie du réseau logique proche d'un graphe aléatoire.

Les nœuds maintiennent des listes de voisins composées de structures contenant notamment l'identifiant de chaque voisin. Nous appelons ces structures "liens" car elles représentent les liens logique reliant deux voisins entre eux. Pour chaque nœud, une limite supérieure (l) est imposée sur la taille de la liste de voisins. Cette limite est d'abord initialisée en fonction d'une estimation de la taille du réseau (elle doit être supérieure au logarithme du nombre de nœuds présents dans le réseau). Ensuite, pendant l'exécution, cette limite peut être repoussée si les ressources physiques du nœud le permettent.

Les listes de voisins sont composées de deux types de lien : *les liens non-applicatifs* et *les liens applicatifs*. La figure A.2 représente la liste de voisins d'un nœud.

Liens non-applicatifs. *Les liens non-applicatifs* sont responsables de maintenir le réseau logique global proche d'un graphe aléatoire avec un faible degré de regroupement (*clustering degree* en anglais). Si l'application est dans un état qui ne nécessite pas de regroupement, par exemple lors de son initialisation, les listes de voisins ne vont contenir que des liens de type non-applicatif. Rappelons que les nœuds n'ont pas besoin d'une connaissance globale et que le nombre de liens non-applicatifs peut varier d'un nœud à l'autre, par exemple en fonction de leur ressources.

Liens applicatifs. Afin de regrouper ensemble les nœuds qui appartiennent à un groupe i, chaque membre du groupe i crée k_i *liens applicatifs* vers d'autres membres du groupe i choisis de manière aléatoire. Ce regroupement favorisera les propagations de mises à jour entre les membres du groupe. Cela va également favoriser la diffusion efficace de messages au niveau

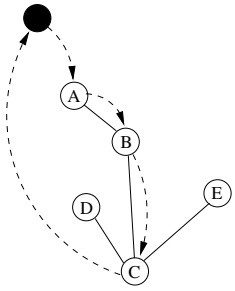

FIG. A.3 – Mécanisme de connexion.

applicatif. Le paramètre k_i est déterminé par l'application et il doit être au moins $\lceil \log(R_i) \rceil$, où R_i est le nombre de membres du groupe i. Le but est de créer un graphe fortement connecté spécifique au groupe i, c'est-à-dire avec une petite longueur de chemin caractéristique (*characteristic path length*). La longueur de chemin caractéristique d'un groupe est la moyenne des longueurs de plus court chemin entre chaque paire de nœuds du groupe.

Politique de remplacement. Quand un lien applicatif pour le groupe i pointant sur le nœud n doit être créé, il sera ajouté à la liste de voisins selon quatre manières différentes.

1. Si la liste de voisins n'a pas atteint la limite supérieure l, et qu'il n'existe pas déjà de lien non-applicatif pointant vers n, un nouveau lien sera ajouté à la liste.
2. Si la limite supérieure de la liste de voisins a été atteinte mais que le nœud a suffisamment de ressources disponibles pour maintenir une liste de voisins plus grande, il peut augmenter la limite supérieure l afin de pouvoir ajouter le lien.
3. Si le nœud ne peut pas augmenter la taille de sa liste, dans ce cas un lien non-applicatif choisi aléatoirement sera supprimé et le lien applicatif ajouté.
4. Enfin, s'il existe un lien non-applicatif pointant vers n, dans ce cas il est transformé en lien applicatif.

A.2.1 Conserver la connectivité du réseau

Le compromis entre les bonnes propriétés des graphes aléatoires et celles apportées en favorisant le regroupement de nœuds peut être réglé en ajustant certains paramètres. Le premier est la taille maximale de la liste de voisins l sur chaque nœud, qui est maintenue comme expliqué ci-dessus. La deuxième est k_i, qui limite le nombre de liens applicatifs qui participent au groupe i. Si $l - \sum_i k_i$ est suffisamment grand (en pratique quelques dizaines pour l'échelle que nous visons), c'est-à-dire que la liste de voisins contient suffisamment de liens non-applicatifs, les bonnes propriétés des graphes aléatoires sont maintenues malgré les regroupements de nœuds créés en prenant l'application en compte. D'une autre côté, il est important de remarquer que $\sum_i k_i$ peut-être plus grand que le nombre de *liens applicatifs*. Cela est dû au fait que certains liens applicatifs peuvent être partagés par de multiples groupes quand les intersections des groupes sont non vides.

Un nouveau nœud se connecte sur le réseau logique (par exemple le nœud noir sur la figure A.3) en contactant un des membres du réseau. Si le nœud qui reçoit la requête de connexion a suffisamment d'espace libre dans sa liste de voisins, il va répondre en envoyant sa liste de voisins courante et ajoutera un lien non-applicatif pointant sur le nouveau nœud. Si la taille de sa liste de voisins a atteint la limite supérieure (ce qui est le cas pour les nœuds *A* et *B* sur la figure A.3), la requête de connexion est retransmise à un voisin choisi aléatoirement. La retransmission de la requête de connexion est associée à une durée de vie maximum en nombre de saut (*Time To Live* ou TTL). Si tous les nœuds qui reçoivent la requête de connexion ont une liste de voisins pleine, le TTL va atteindre 0 et le dernier nœud ayant reçu la requête de connexion sera forcé d'ajouter un lien non-applicatif vers le nouveau nœud (en remplaçant un lien non-applicatif). Le nœud acceptant la requête répond en envoyant sa liste de voisins (le nœud *C* sur la figure A.3). Le nouveau nœud utilise ensuite la liste de voisins reçue pour créer sa propre liste.

Le protocole de détection de fautes est basé sur les protocoles de SWIM (pour *Scalable Weakly-consistent Infection-style process group Membership* [41]). Le temps est divisé en périodes de durée T secondes. Chaque nœud envoie à chaque période un message "ping" à l'un de ses voisins. Le nœud cible est choisi en parcourant un tableau représentant une permutation aléatoire de la liste de voisins. Chaque fois que le tableau est parcouru complètement, une nouvelle permutation aléatoire est calculée. Le nœud attend une réponse au message "ping" avec un délai de garde de t ($< T$) secondes. Si la réponse n'est pas reçue à temps, un message "ping" indirect est envoyé à y nœuds. Ces y nœuds vont envoyer un message "ping" au nœud initialement sélectionné et s'ils reçoivent une réponse, cette dernière est retransmise au nœud initiateur du message "ping". Les messages "ping" indirects peuvent permettre de contourner des problèmes réseaux temporaires. En revanche, si aucune réponse n'est reçue avant la fin de la prochaine période du protocole, le nœud cible est suspecté d'être défaillant. Au début de chaque période, tous les nœuds qui ont été suspectés sont supprimés de la liste de voisins.

De manière à obtenir un réseau logique avec un faible coefficient de regroupement et une bonne distribution des degrés entrant sur les nœuds, toutes les U périodes de protocole chaque nœud vérifie si la liste de ses liens non-applicatifs a été modifiée. S'il n'y a pas eu de modifications après U périodes, il envoie un message de connexion à un nœud choisi aléatoirement. Avec la liste de voisins reçue en réponse, le nœud va essayer de remplacer une fraction[2] choisie aléatoirement de sa liste de voisins. Remarquons que plus U est petit, plus le remplacement sera significatif et plus vite le réseau logique tendra vers un coefficient de regroupement faible et stable.

A.2.2 Communication de groupe

Comme expliqué précédemment, quand un nouveau lien applicatif est créé, il en résulte une suppression d'un lien non-applicatif excepté si le nœud a suffisamment de ressources pour agrandir la taille de sa liste de voisins. De cette manière, nous gardons sur chaque nœud presque constant le coût de la maintenance des listes de voisin. De plus, un lien applicatif peut être partagé. Par exemple, considérons un nœud a qui appartient au groupes i et j.

[2] $f = 50\%$ dans nos évaluations.

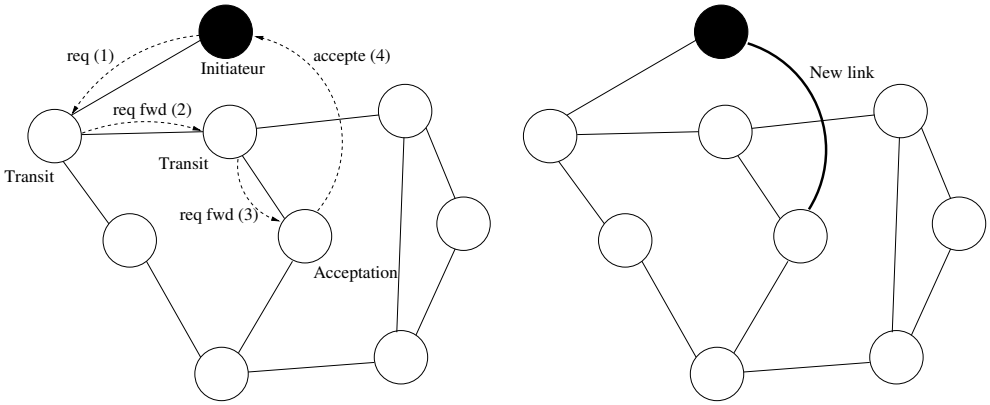

FIG. A.4 – Le mécanisme de marche aléatoire.

Si le nœud b se connecte aux groupes i et j, un seul lien applicatif est créé avec le nœud a sachant que ce lien est *partagé*. Un compte du nombre de partages est tenu à jour pour les liens partagés.

Marche aléatoire pour les liens applicatifs. Afin de faire face à la nature dynamique de l'infrastructure et pour éviter des topologies pathologiques (chaîne, anneau, étoile, etc.) qui peuvent être induites par les occurrences de fautes, il est important de *rafraîchir* les liens périodiquement. C'est également utile afin de garantir un court chemin entre deux nœuds d'un groupe donné. Ajouter $O(n)$ liens non-applicatifs à un graphe possédant n sommets réduit le diamètre du graphe à $O(\log(n))$ [21, 104]. Ce résultat ne s'applique qu'à des graphes bidirectionnels. Aussi nous imposons que tous les liens applicatifs soient bidirectionnels. Quand un lien est créé de a à b, b ajoute également un lien vers a. Si le nœud b supprime le lien alors le nœud a le supprime également. Quand un lien applicatif est partagé, il est maintenu tant que le nombre de partages est strictement positif. Quand un lien applicatif cesse d'être utilisé en tant que tel, il se transforme en lien non-applicatif.

Le graphe est rafraîchi périodiquement par l'exécution des points suivants par chaque nœud d'un groupe applicatif :

1. lancer une marche aléatoire pour découvrir un nouveau voisin. La marche aléatoire ne traverse pas plus de TTL nœuds en utilisant les liens applicatifs associés au groupe ;
2. supprimer un ancien lien quand le nouveau est créé.

Bien que le délai de garde entre chaque marche aléatoire soit un paramètre de l'application, il est important de noter que les nœuds appartenant à un groupe ne sont pas synchronisés.

A.3 Évaluation du réseau logique pair-à-pair MOVE

Nous avons utilisé un simulateur à événements discrets décrit à la section A.3.1 afin d'évaluer notre réseau pair-à-pair malléable.

A.3.1 Simulation par événements discrets

Afin d'évaluer le réseau pair-à-pair malléable MOVE (pour l'anglais *Malleable OVEerlay*), nous avons choisi de mettre en œuvre un simulateur à événements discrets : MOVESIM. Ce simulateur comprend 5000 lignes de code Java.

Réalisme du simulateur. MOVESIM peut prendre en paramètre des topologies réseaux générées par GT-ITM [25], un générateur reconnu pour sa capacité de produire des topologies réalistes. Des traces d'applications peuvent également être prises en compte. Ces deux points permettent d'augmenter le réalisme du modèle simulé et ainsi de rendre les simulations plus fidèles à la réalité.

But du simulateur. Le but est de simuler le comportement d'un réseau logique pair-à-pair. Nous nous focalisons sur la gestion des listes de voisins. Le but est de "rapprocher" les nœuds susceptibles d'avoir de nombreuses communications en réduisant le nombre de "sauts" qui les séparent. Les groupes de nœuds communicant fréquemment sont appelés groupes applicatifs. Ce simulateur reproduit donc le comportement de chacun des nœuds en exécutant nos algorithmes de gestion de listes de voisins. En sortie, une trace d'exécution est générée ainsi que les matrices représentant les liens entre les nœuds pour chacun des groupes applicatifs gérés par le réseau malléable. Cela permet d'évaluer la connectivité de chaque groupe applicatif, en présence ou non de fautes.

Fonctionnement. À l'initialisation : 1) une matrice décrivant les latences/débits réseau entre les nœuds est créée à partir de la topologie ; 2) une matrice contenant l'état initial de chacun des nœuds est également calculée ; et 3) des fichiers de configuration et des fichiers de traces sont utilisés pour générer les événements applicatifs. Ces événements correspondent aux arrivées et aux départs de nœuds dans le réseau pair-à-pair, aux créations par des nœuds de groupes applicatifs, aux arrivées et aux départs de nœuds de groupes applicatifs, aux diffusions de messages au sein des groupes applicatifs, et aux déclenchements de délais de garde. Tous ces événements sont composés : 1) d'une action à exécuter, 2) de l'identifiant du nœud devant exécuter l'action, et 3) d'une date d'exécution. Ils sont insérés dans une file d'événements triée selon la date.

Le simulateur itère ensuite en traitant un événement de la file à chaque itération. Pour chaque événement, le traitement est le suivant.

1. La date de simulation est avancée à la date de l'événement courant.
2. Le simulateur charge l'état du nœud contenu dans l'événement.
3. L'action de l'événement est exécutée.

Chaque action exécutée peut modifier l'état du nœud (sa liste de voisins par exemple) et générer de nouveaux événements dans la file. Les événements générés en cours d'exécution sont des réceptions de messages ou des déclenchements de délais de garde.

Quand le temps alloué à la simulation s'est écoulé[3], les listes de voisins sont analysées afin de calculer les matrices de voisinage et les *longueurs de chemin caractéristiques* de chaque groupe applicatif.

[3]La durée de simulation est donnée en paramètre.

L'ensemble des évaluations présentées ici ont été réalisées avec ce simulateur à événements discrets. Le but de ces simulations est d'illustrer comment le réseau logique MOVE *s'adapte* aux applications, de mesurer la connectivité entre les nœuds appartenant à un même groupe applicatif, et enfin d'évaluer la résistance du réseau logique face aux fautes.

Les simulations effectuées avec moins de 2000 nœuds utilisent des topologies générées par le générateur GT-ITM, celles utilisant 2000 nœuds sont réalisées avec une topologie plate, c'est-à-dire une latence et un débit identique entre chaque nœud.

A.3.2 Adaptation du réseau logique.

Pour analyser comment le réseau logique réagit face aux besoins des applications, nous lançons des simulations sur un réseau de 520 nœuds en variant le nombre de groupes créés par les applications. Le paramètre k, correspondant au nombre de liens applicatifs qu'un nœud essaye de conserver pour chaque groupe, est fixé à 5 pour chacun des groupes sur chacun des membres. Nous observons sur la figure A.5 que le nombre de liens total ne change

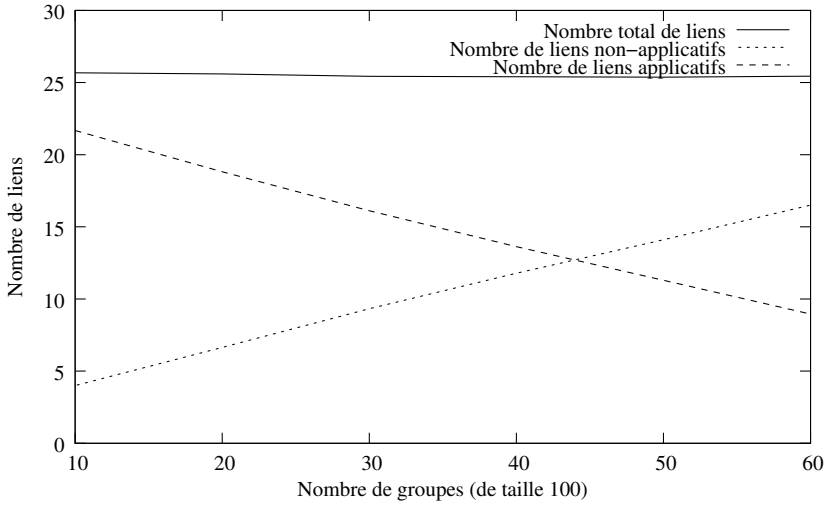

FIG. A.5 – Nombre de liens applicatifs et non-applicatifs dans le réseau logique en fonction du nombre de groupes.

pas mais que les liens non-applicatifs sont progressivement remplacés par des liens applicatifs. MOVE est donc bien un réseau malléable qui adapte sa structure aux applications.

A.3.3 Partage de liens applicatifs.

La transformation de liens non-applicatifs en liens applicatifs illustrée à la section précédente augmente le coefficient de groupement du réseau. Les liens non-applicatifs sont responsables du maintient du réseau global, c'est-à-dire de la connectivité entre les nœuds indépendamment de leur appartenance à un groupe applicatif. Afin de conserver un nombre maximal de liens non-applicatifs, les liens applicatifs sont partagés lorsque les intersections des groupes sont non-vides. Nous simulons un réseau de 520 nœuds dans lequel nous créons 60 groupes applicatifs de taille 100 et de paramètre k égal à 5. À des fin de lisibilité, seuls

100 nœuds sont illustrés sur la figure A.6. Chaque point de l'axe des abscisses représente un nœud et la différence entre les deux courbes représente le gain réalisé grâce au partage.

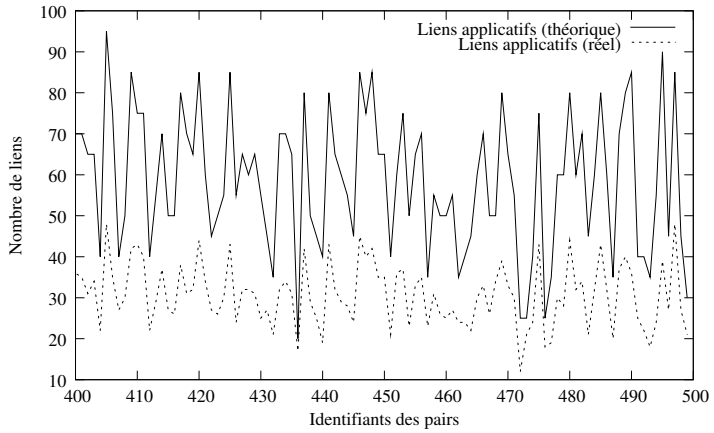

FIG. A.6 – Nombre de liens applicatifs sur chaque nœud.

A.3.4 Connectivité au sein des groupes.

Le but premier de notre approche est de permettre aux nœuds communiquant fréquemment d'être proches au sein du réseau logique. Afin d'évaluer cette connectivité nous simulons un réseau de 2000 nœuds au sein duquel sont créés 50 groupes dont les tailles correspondent à des *slices*[4] de PlanetLab [124]. Pour chaque groupe, le paramètre k est égal au logarithme de la taille du groupe. La figure A.7 présente les mesures de longueur de chemin

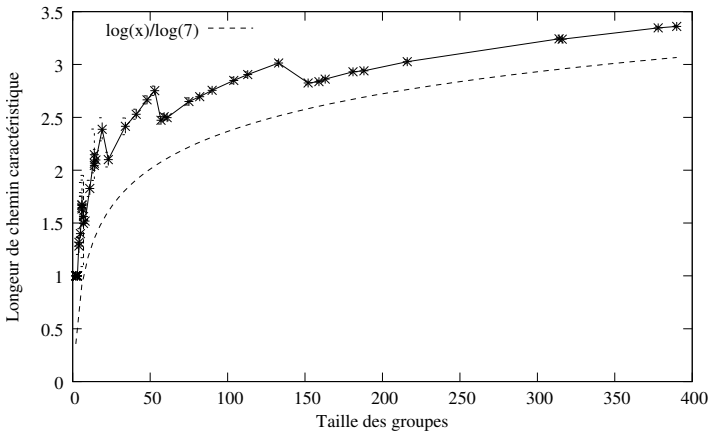

FIG. A.7 – Longueur de chemin caractéristique pour 50 Groupes PlanetLab.

caractéristique en fonction de la taille des groupes. Ici, cette longueur reste limitée, elle reste inférieure à 3.5 pour des groupes de 400 membres. Des simulations effectuées avec des tailles de groupes aléatoires donnent des résultats équivalents.

[4] une *slice* correspond à un ensemble de nœuds virtuels utilisé par une application ou un service.

A.3.5 Tolérance aux fautes.

Malgré les groupements de nœuds réalisés pour améliorer les communications des applications, le réseau logique doit rester connecté. Pour évaluer la tolérance à un grand nombre de fautes simultanées, nous avons configuré les nœuds pour être défaillants (défaillance franche, ou crash) avec une probabilité p après 2 heures de temps de simulation. Les simulations ont été effectuées sur un réseau de 2000 nœuds comprenant 50 groupes dont les tailles sont distribuées exponentiellement, et dont les paramètres k sont égaux aux logarithmes des tailles. Avec cette configuration, le réseau logique reste connecté tant que la probabilité p est inférieure à 0.5, il supporte donc bien un grand nombre de fautes simultanées.

D'autres évaluations de MOVE sont présentées dans [MMAG06].

A.4 Analyse

Le réseau logique est très important pour les applications qui l'utilisent. Toutes leur communications seront acheminées par lui. Il nous semble donc important que ce réseau prenne en compte les besoins des applications en terme de communication.

Au cours de cette collaboration, nous avons conçu le réseau logique MOVE que les applications peuvent influencer afin qu'il s'adapte aux schémas de communications et ainsi obtenir des communications performantes.

L'évaluation de ce réseau logique montre notamment qu'il est possible d'améliorer les communications au sein des groupes applicatifs tout en offrant une bonne tolérance à la volatilité.

Il est intéressant de remarquer que l'on peu utiliser la notion de groupes applicatifs décrite dans cette annexe pour implémenter des groupes de réplication d'une donnée. Alors que JUXMEM offre des modèles de cohérence forte en s'appuyant sur la hiérarchie, MOVE peut offrir des modèles de cohérence plus relâchés en se basant sur une approche probabiliste. Il est difficile de dire qu'une approche est meilleure que l'autre, en effet il semble qu'une approche de type MOVE peut viser une échelle un peu plus grande sur des réseaux de performance moindre, cependant le prix à payer est un relâchement des contraintes de cohérence des données, les applications visées ne sont donc pas les mêmes.

JUXMEM est adapté à des applications de type scientifique (comme du calcul matriciel), sur des grilles de calcul. MOVE semble plus adapté à des applications collaboratives sur un réseau de type Internet.

Sébastien MONNET

Gestion des données dans les grilles de calcul : Support pour la tolérance aux fautes et la cohérence des données.

Mots-clefs : Grille de calcul, gestion de données,
tolérance aux fautes, cohérence de données, approche pair-à-pair,
protocoles hiérarchiques.

Les applications scientifiques d'aujourd'hui, telles les simulations de grands phénomènes naturels, requièrent une grande puissance de calcul ainsi qu'une importante capacité de stockage. Les *grilles de calcul* apparaissent comme une solution permettant d'atteindre cette puissance par la mise en commun de ressources de différentes organisations. Ces architectures présentent en revanche des caractéristiques rendant leur programmation complexe: elles sont *dynamiques*, *hétérogènes*, réparties à *grande échelle*.

Cette thèse s'intéresse aux problématiques liées à la conception d'un service de partage de données pour les grilles de calcul. L'objectif est de permettre un accès *transparent* aux données, en automatisant la *localisation*, le *transfert*, la gestion de la *persistance* et de la *cohérence* des données partagées.

Nous nous sommes plus particulièrement concentrés sur la gestion de la *cohérence* et de la *persistance* des données en environnement dynamique. Dans un tel contexte, assurer la persistance nécessite la mise en place de mécanismes de *tolérance aux fautes*. Nous proposons une approche pour gérer *conjointement* ces deux aspects via une architecture logicielle multi-protocole permettant de coupler différents protocoles de cohérence avec différents mécanismes de tolérance aux fautes.

Nous proposons une conception *hiérarchique* de cette architecture, adaptée à la topologie réseau des grilles de calcul. Ces contributions ont été mises en œuvre au sein du service de partage de données pour grilles JUXMEM. Les expérimentations menées sur la grille expérimentale Grid'5000 montrent que notre conception hiérarchique permet d'améliorer les performances des accès aux données partagées.